**MASTER EN SCIENCES INFORMATIQUES**

# Infrastructure pour la reconfiguration dynamique des logiciels de contrôle pour satellites

Vleminckx, Benoît

*Award date:*
2008

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique

*Année Académique 2007 - 2008*

# Infrastructure pour la reconfiguration dynamique des logiciels de contrôle pour satellites

Benoît Vleminckx


Mémoire présenté en vue de l'obtention du grade de
Maître en Sciences Informatiques

**Résumé**

Lors de la phase de maintenance, les logiciels nécessitent souvent des modifications (correction de bug, ajout de fonctionnalité, etc.) La plupart du temps, le logiciel peut être arrêté, modifié, recompilé et redémarré. Cependant, dans quelques domaines particuliers (caches web, systèmes embarqués, systèmes autonomes, applications critiques, etc.), le système ne peut être arrêté. La reconfiguration dynamique désigne la capacité pour un logiciel de transformer son propre code source, sa structure ou son implémentation, sans être arrêté et en demeurant dans un état consistant.

mots-clefs : reconfiguration dynamique, architectures orientées-composant, domain specific, satellites

**Abstract**

During maintenance phase, software sometimes needs modification (bug fixing, evolution, etc.) In most cases, the software could be halted, modified, recompiled and restarted. However, in some other domains (web caches, embedded systems, autonomous systems, critical applications, etc.), the system can not be stopped. Dynamic reconfiguration is the ability for a software to transform its own source code, structure, or implementation, without being stopped and so maintaining a consistent state.

keywords : dynamic reconfiguration, component-oriented architectures, domain specific, satellites

# Remerciements

Je tiens à exprimer ma reconnaissance à mon promoteur Vincent Englebert pour sa patience et ses corrections, ainsi qu'à mon maître de stage Fabien Dagnat, assisté par Jérémy Buisson, pour m'avoir reçu dans leur équipe et guidé dans mon travail. Je tiens également à remercier les autres membres du personnel de Telecom Bretagne avec qui j'ai eu la chance de pouvoir m'entretenir et qui m'ont beaucoup appris. D'une façon générale, ce document est dédié à toutes les personnes qui m'ont rendu le séjour agréable et avec qui mes rapports furent aussi divers qu'enrichissants.

# Table des matières

# Table des figures

# Glossaire

| | |
|---|---|
| ACID | Atomicité, Cohérence, Isolation, Durabilité |
| AOP | Aspect-Oriented Programming |
| API | Application programming interface |
| AOSD | Aspect-Oriented Software Developpement |
| CF | Component framework |
| DHS | Data Handling System |
| DLL | Dynamic Link Librairy |
| DSMM | Domain-Specific Metamodel |
| DSML | Domain-Specific Meta-Language |
| DSL | Domain-Specific Language |
| EEPROM | Electrically-Erasable Programmable Read-Only Memory |
| FDIR | Failure Detection, Isolation and Recovery |
| JVM | Java Virtual Machine |
| JVMS | Java Virtual Machine Specification |
| LOC | Line of Code |
| LV | Logiciel de Vol |
| MIPS | Million d'instructions par seconde |
| OBSW | On Board Software |
| OOP | Object-Oriented Programming |
| SCAO (ou AOSC) | Système de Contrôle d'Attitude et d'Orbite |
| PUS | Packet Utilization Standard |
| RTOS | Real-Time Operating System |
| TC | Télécommande |
| TM | Télémesure |

# Chapitre 1

# Introduction

L'évolutivité d'un système est l'une des caractéristiques non fonctionnelles les plus importantes. Elle désigne la capacité et la facilité avec lesquelles un système peut être transformé, corrigé, maintenu. Une évolution dans un système d'information typique peut consister en l'ajout ou la modification d'un ou plusieurs modules couvrant les nouveaux besoins de l'organisation qui l'utilise.

Dans la plupart des cas, le système peut être arrêté, modifié, recompilé et remis en exploitation. Cependant, dans plusieurs domaines particuliers tels que les caches web (voir section 3.2), les systèmes embarqués, les systèmes autonomes, les agents électroniques, etc. le système ne peut être arrêté. Il doit être modifié sans interruption et en demeurant bien entendu dans un état cohérent. On parle alors de **reconfiguration dynamique**, qu'on désigne parfois par *hotswapping* ou *runtime evolution*. On parle également de modifications « à chaud », « à la volée » ou « *on the fly* ».

Les logiciels de vol des satellites nécessitent souvent des évolutions importantes en cours d'exploitation (correction de bugs, patch de données techniques, maintenance, ajout ou suppression[1] de fonctionnalité, etc.), rendues encore plus difficiles par la situation de l'astronef en vol. Actuellement, les techniques d'évolution sont peu pratiques et ont généralement lieu sans prendre en compte la structure du logiciel, à l'exemple de la technique du *bindiff* [32].

Cet outil compare deux versions d'un fichier binaire et détecte les différences. Ce sont ces différences qui, sous forme de script, sont transmises

[1] Il peut être nécessaire de supprimer certaines fonctionnalités, potentiellement dangereuses après les avoir utilisées.

aux satellites afin d'actualiser le logiciel. Cette méthode nécessite cependant de connaître une liste des points d'accès dans le logiciel (qui s'élève pour l'instant à plusieurs milliers), de gérer tant bien que mal l'ensemble de ces modifications et d'effectuer les sauts nécessaires à leur exécution[2]. Elle exige également de gérer une liste des patchs exécutés dans les différents satellites d'une même famille (souvent appelée constellation). En effet, on imagine que si pour la correction d'un bug, le même patch peut être appliqué indifféremment à tous les logiciels de vol d'une même famille, d'autres types de modifications dépendent de problèmes matériels (une modification peut consister à « isoler » un composant matériel défectueux et à le remplacer par le composant redondant prévu à cet effet) et sont particuliers à chaque logiciel de vol en exécution. Ce qui rend la gestion des différentes versions et les évolutions ultérieures plus difficiles encore, puisque chaque patch est écrit en fonction des modifications précédentes et de l'état du logiciel.

Faut-il également préciser que le faible débit entre la terre et le satellite en vol (au maximum de 4 ko/s) limite la taille des patchs et la portée des modifications. De plus, si beaucoup de satellites sont géostationnaires (ils possèdent une période de révolution exactement égale à la période de rotation de la Terre, de telle façon qu'ils semblent immobiles par rapport au sol), d'autres sont en mouvement par rapport au sol. On ne peut communiquer avec ceux-ci que par périodes.

La complexité croissante (évolution de quelques milliers de LOC dans les années 80 à plusieurs centaines de milliers actuellement [39]) des logiciels de vol ainsi que l'émergence des systèmes distribués entre plusieurs satellites d'une même constellation rendent le développement et la maintenance des logiciels de vol de plus en plus difficiles. Le projet SP@CIFY a pour but de créer un *framework* facilitant le développement des logiciels de vol. Ce framework devra inclure des mécanismes permettant d'effectuer dynamiquement les modifications nécessaires, avec facilité pour les développeurs du logiciel de base ainsi que pour les développeurs chargés des évolutions. Dans ce contexte et au vu des récentes améliorations des architectures logicielles (comme les modèles orientés-composant), ces approches de reconfiguration dites *memory dump and bit patching* deviennent obsolètes et fort peu pratiques. Les approches à base de composant devraient permettre de réduire la complexité.

[2]Afin de minimiser la taille des patchs, les modifications les plus simples sont écrites directement en langage assembleur, ce qui implique de devoir gérer manuellement les flots d'exécution du logiciel.

Le stage, qui s'est déroulé entre septembre 2007 et janvier 2008, a eu pour objet principal l'étude de solutions de reconfiguration dynamique existantes afin d'y déceler les difficultés rencontrées et d'analyser les différents protocoles de reconfiguration envisagés. Cette étude de l'état de l'art (décrite et recontextualisée ici dans le chapitre 3) a contribué modestement au développement d'une nouvelle approche (décrite dans le chapitre 4), envisageant le problème sous un angle plus abstrait et qui réponde plus précisément aux besoins de l'industrie des satellites. La seconde partie du stage a consisté à écrire une implémentation de cette approche. Le document est structuré comme suit.

Le chapitre 2 explique plus en détails les spécificités matérielles et logicielles d'un satellite et du système embarqué, ainsi que les objectifs du projet SP@CIFY.

Le chapitre 3 propose d'étudier différentes solutions fournissant des capacités de reconfiguration dynamique, à travers différents paradigmes de programmation (programmation impérative, orientée-objet, orientée-composant et orientée-aspect). Ce chapitre permettra de découvrir les problèmes qui se posent dans chacune de ces solutions et d'introduire au fur et à mesure les concepts nécessaires à la compréhension du chapitre 4.

Enfin, le chapitre 4 propose une approche générale pour la reconfiguration dynamique, inspirée notamment par les problèmes soulevés dans le chapitre sur l'état de l'art. Le chapitre 5 tentera de comparer cette dernière approche à celles étudiées dans l'état de l'art, d'analyser ses avantages et ses inconvénients et d'apporter des débuts de solutions aux questions restées en suspend.

# Chapitre 2

# Contexte

## 2.1 Architecture matérielle d'un satellite

Le système satellite est un engin spatial sans pilote dont l'architecture est fortement caractérisée par la mission à accomplir [39]. On peut néanmoins affirmer qu'il comprend généralement deux sous-systèmes principaux, souvent séparés matériellement, qui sont la **plate-forme** (en orange sur la figure 2.1) et la **charge utile** (en bleu). Cette dernière contient une variété d'instruments scientifiques qui dépend de la mission du satellite. Il peut s'agir de télescopes, d'appareils photographiques, d'équipements destinés à assurer des télécommunications, etc.

La plate-forme, de son côté, gère le satellite en tant que véhicule. Elle comprend la structure mécanique du satellite dont le matériel nécessaire pour utiliser la charge utile : communications sol/bord, pointage, régulation thermique, régulation de la puissance électrique, etc.

On peut également distinguer l'**avionique**, qui inclut l'ensemble des éléments matériels et logiciels embarqués de la plate-forme qui permettent de contrôler le satellite. On y trouve notamment (selon un premier niveau de décomposition) :

- Les interfaces de télémesure et de télécommande (TM/TC) assurant la communication sol/bord. Les télémesures permettent d'obtenir des informations du satellite, tandis que les télécommandes permettent de lui envoyer des ordres
- L'interface avec la charge utile, qui permet par exemple de recevoir les données de télémesure

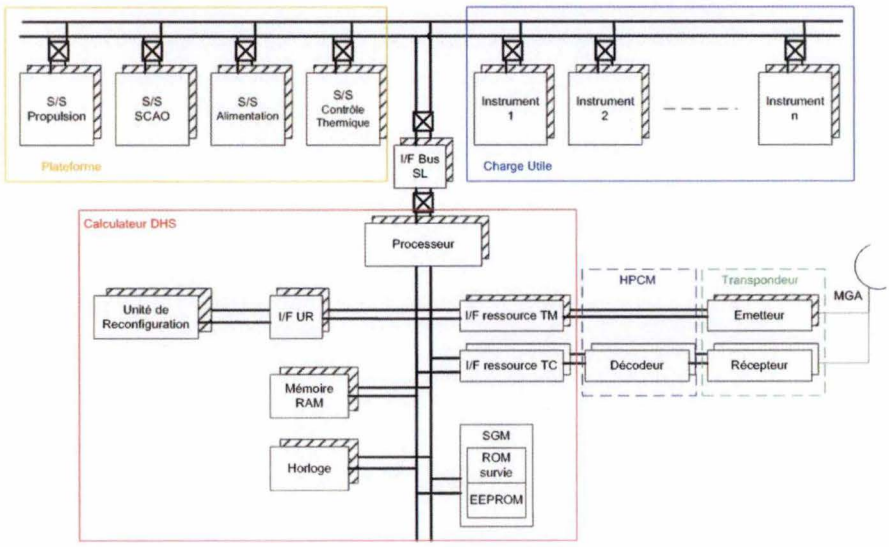


FIG. 2.1 – Architecture matérielle de référence d'un satellite. Les composants affichés en double sont redondés. Source : [39]

- Le contrôle thermique
- Le contrôle énergétique
- Le Data Handling System (DHS) qui fournit un certain nombre de services logiciels de bas niveau : gestion des interfaces sur le bus interne, gestion des entrées/sorties, la gestion du format des données utilisées dans les TM/TC, etc.
- Le Système de Contrôle d'Attitude et d'Orbite (SCAO ou AOCS) qui gère la position et l'orientation du satellite dans l'espace[1]
- Un sous-système de House Keeping (HK) qui fournit certaines informations sur l'état de santé du système
- Un sous-système de Fault Detection, Isolation et Recovery (FDIR) qui gère la surveillance et la configuration de l'équipement

Le matériel contenu dans un satellite est exposé à plusieurs types d'agressions : rayonnement, température, etc. qui expliquent la présence des sous-systèmes de HK et de FDIR cités plus haut, ainsi que la redondance de plusieurs composants matériels et logiciels.

[1]L'attitude d'un satellite, et non pas l'altitude, désigne l'orientation d'un repère tridimensionnel lié au satellite par rapport à un autre repère.

Ces sous-systèmes peuvent être eux-mêmes analysés et décomposés. Dans le SCAO par exemple, on trouve les sous-systèmes suivants :

- Le Sous-système de Détermination d'Attitude et d'Orbite,
- Un sous-système FDIR (matériel et logiciel)
- Plusieurs types d'actionneurs (propulseurs qui expulsent de la masse pour se déplacer, roues à réaction dont on utilise l'inertie afin de s'orienter avec précision)
- Plusieurs types de capteurs (senseurs Terre/Soleil, magnétomètres qui mesurent le champ gravitationnel de la Terre, senseurs stellaires - ou *star trackers* - qui mesurent précisément l'altitude par rapport à une cartographie céleste, etc.), ainsi que les interfaces matérielles et logicielles de tous ces équipements.

Il faut également remarquer la faible puissance du matériel embarqué. Pour donner un ordre de grandeur : 8 mo de RAM, 6 mo d'EEPROM et 15 MIPS[2].

## 2.2 Architecture logicielle d'un satellite

On peut décomposer l'architecture logicielle d'un satellite en quatre couches principales (cf. figure 2.2) :

- La couche Mission et Système donne accès aux fonctionnalités logicielles de plus haut niveau.

- La couche applicative fournit l'accès aux données associées aux différents composant matériels (dans l'ordre, SCAO, contrôle énergétique, contrôle thermique, télécommande et télémesure, interface avec la plateforme).

- La couche DHS fournit des services logiciels de base. Le PUS (pour Packet Utilization Standard) est un standard de communication qui définit l'interface sol/bord au niveau applicatif [33].

- La couche RTOS (pour *Real-Time Operating System*) offre les services d'un système d'exploitation qui prend en compte les contraintes temps-réel du satellite. Le BSP (*Board Support Package*), commun dans les systèmes embarqués, est un support logiciel de bas niveau qui assure la

[2]Comparé aux 300 MIPS de certains téléphones portables.

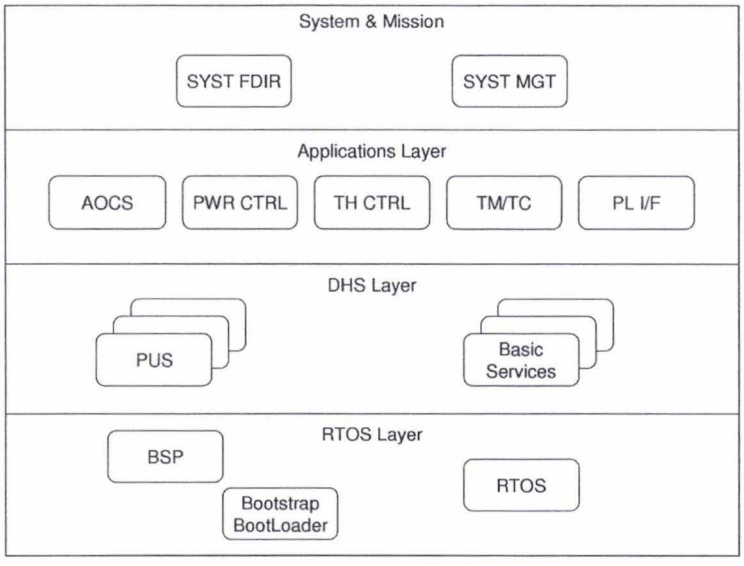


FIG. 2.2 – Architecture logicielle typique d'un satellite

liaison entre le système d'exploitation et la carte mère. Ce logiciel est lié au bootstrap qui contient le support matériel minimum pour charger le système d'exploitation et les pilotes nécessaires pour l'utilisation du matériel.

## 2.3 Le projet SP@CIFY

Le but global du projet SP@CIFY est de simplifier, d'améliorer la qualité et diminuer les coûts de développement et de maintenance des logiciels de vol, en synthétisant l'expérience accumulée dans ce domaine [6, 7]. En effet, le développement des logiciels de vol, qui incluent des contraintes matérielles, de temps-réel en plus des exigences de reconfiguration, est fort complexe, et est appelé à encore augmenter d'un échelon de complexité, avec l'ajout continu de nouvelles fonctionnalités et le développement de logiciels répartis dans une même constellation. Le projet vise à concevoir un atelier de développement et de maintenance de logiciels de vol. Le caractère exploratoire du projet permettra d'aboutir à des techniques assez génériques pour être appliquées aux domaines subissant des contraintes similaires.

Le projet SP@CIFY devra spécifier un intergiciel (ou *middleware*) permettant de communiquer facilement entre les différents sous-systèmes logi-

ciels et matériels évoqués ci-dessus, et à terme, entre les différents noeuds d'une même constellation. La plupart des intergiciels actuels sont souvent lourds et lents, alors que le matériel embarqué dans les satellites est particulièrement peu puissant et que le système doit répondre à des contraintes de temps-réel. Il existe certains intergiciels qui répondent aux attentes des logiciels embarqués, tel que CORBA/e. Mais on trouve peu d'intergiciels qui s'attaquent à la fois aux problèmes posés en matière de contraintes physiques propres aux systèmes embarqués, aux contraintes temps-réel et aux besoins de reconfiguration dynamique.

Concrètement, le projet devrait aboutir à la création d'un *framework*, ce qui signifie ici, une base logicielle, un logiciel à moitié terminé, qui fournirait un certain nombre de services. Il s'agirait de la partie de l'intergiciel configuré à partir des modèles représentant l'application. Ce *framework* serait instancié et spécifié pour un projet donné.

# Chapitre 3

# Reconfiguration Dynamique : État de l'art

Le problème de la reconfiguration dynamique n'est pas nouveau. Un article de Faber datant de 1976 y fait déjà référence [22]. La volonté de pouvoir transformer dynamiquement un système a depuis lors donné le jour à plusieurs approches très différentes, et s'est exprimée à travers de nombreux paradigmes de programmation. Bon nombre de ces solutions sont peu abstraites et proches du code source de l'application à modifier. Parmi celles-ci, on compte Arachne (ou $\mu$Dyner) qui se base sur la programmation orientée-aspect [36, 37, 35, 38, 34, 20, 23, 31, 30, 8, 28]. Nous verrons également les mécanismes qui ont été ajoutés *a posteriori* au langage Java et qui permettent un certain nombre de modifications à la volée [17, 19, 18, 24, 29].

Ces deux solutions ressemblent à bien des égards aux techniques actuellement utilisées dans les domaines des logiciels de vol et leurs inconvénients sont similaires. Elles sont entre autres difficiles à utiliser dans des systèmes très complexes. Cependant, il existe maintenant des modèles architecturaux plus abstraits, comme les modèles orientés-composant dont certains proposent des opérations permettant d'effectuer des reconfigurations dynamiques. Nous en étudierons deux : Fractal [4, 9, 14, 15] et OpenCOM [11, 10, 25, 12, 1, 26, 2, 13].

Puisque la plate-forme de développement SP@CIFY devra être plus abstraite et plus simple d'emploi que les techniques de développement et de reconfiguration actuelles, nous constaterons que ces deux derniers modèles sont plus proches de la solution de reconfiguration à venir.

C'est au terme de notre parcours que nous proposerons une nouvelle solution, en l'adaptant au contexte particulier des logiciels de vol pour satellites et en prenant en compte les problèmes et les difficultés relevés dans ces différentes solutions.

## 3.1 Caractéristiques d'analyse

Afin de voir plus clair dans la diversité des outils, des modèles et des protocoles de reconfiguration dynamique que nous allons aborder, voyons quelques critères d'analyse qui nous permettront de penser plus aisément les différences et les points communs.

**Granularité.** Nous désignerons par granularité le concept permettant de cerner l'unité logique sur laquelle a lieu une opération de reconfiguration dynamique. En effet, dans certains protocoles, il est possible de transformer une fonction ou une méthode. Dans d'autres, on dispose d'un ensemble d'opérations permettant de remplacer un composant. Le terme est explicitement employé par [27, 21, 32], alors que [17] parle d'« unité d'évolution ».

Nous préciserons également les opérations autorisées sur cette unité. Tantôt en effet, on peut *modifier* l'objet. Tantôt, on peut créer de nouvelles instances à partir d'un type. Certains outils permettent uniquement d'ajouter des éléments quand d'autres permettent également d'en supprimer.

**Type de modification.** Ce critère est intimement lié à la granularité. Tandis que la granularité sert à cerner l'élément sur lequel a lieu la modification, ce critère tente d'analyser le paradigme dans lequel a été pensé la reconfiguration. Nous distinguerons les modifications comportementales, impliquant l'ajout ou la suppression d'une portion de code, des modifications structurelles qui impliquent l'ajout, la suppression ou la modification d'une entité représentant une partie de l'application (comme un composant ou une classe) ou la modification du lien qui unit deux entités.

**Ouverture.** Cette caractéristique reflète la capacité de la solution à accepter des modifications dynamiques non prévues lors du design de l'application. Elle se décline en trois niveaux, relevés par [27] : les solutions de reconfiguration peuvent être fermées, partiellement ouvertes ou ouvertes :

- **Fermée** : Toutes les modifications susceptibles d'avoir lieu sur une application à reconfigurer sont prévues à l'avance et codées (voire hard-codées) au moment du développement. Nous n'étudierons ici aucune solution fermée, puisqu'elles n'offrent guère de possibilité pour les modifications non prévues. Elles ne sont pas pertinentes.

- **Partiellement ouverte** : Puisqu'on ne peut prévoir toutes les modifications lors du design d'un système, certains outils permettent de mettre à disposition un ensemble de points d'ouverture, qui pourront être accédés lors de l'exécution afin d'insérer de nouvelles fonctionnalités ou de nouveaux composants, écrits ultérieurement. Les fonctionnalités destinées à être branchées au système original ne sont pas prédéfinies au moment du design du système.

- **Ouverte** : Contrairement aux deux autres niveaux, dans les solutions ouvertes, tous les éléments du système sont vus comme des entités manipulables. C'est ici que l'on classe les systèmes les plus flexibles et les plus adaptables. Précisons qu'une solution très ouverte n'est pas forcément meilleure qu'une solution peu ouverte. En effet, pour plus de sécurité par exemple, on pourrait décider dès la conception du système que certains éléments critiques demeureront inchangés durant toute la vie du logiciel.

**Gestion de l'état courant.** Nous analyserons sous ce critère les différents mécanismes et techniques mis en place afin de conserver un état cohérent. Ceci inclut, par exemple, des mécanismes de transfert d'information entre une ancienne et une nouvelle version d'un même élément remplacé dans le système, où les façons dont sont aiguillés les flots d'exécution lors d'une transformation au niveau du code.

## 3.2 Arachne

Arachne a été développé pour répondre au besoin d'adaptabilité des caches web, plus précisément de Squid. Les caches web utilisent les spécifications des protocoles utilisés sur Internet (tel que HTTP) pour répliquer les pages web demandées par les utilisateurs : lorsqu'un internaute demande une page sur un serveur distant, la requête est d'abord adressée à un cache web, géographiquement plus proche de l'utilisateur que le serveur distant. Si le cache web possède localement une copie de la page, il la renvoie directement à l'utilisateur. Dans le cas contraire, il récupère le document depuis Internet, l'envoie à l'utilisateur et sauvegarde une copie afin de répondre plus efficacement aux prochaines requêtes.

Les caches web ont trois avantages majeurs : tout d'abord, la consommation de bande passante entre les caches web et les serveurs est diminuée. Conséquemment, les serveurs sont moins chargés puisque moins de requêtes leurs sont directement adressées. Enfin, puisqu'ils sont assez nombreux et géographiquement plus proches des utilisateurs que les serveurs distants, ils réduisent le temps de latence observé par les utilisateurs finaux.

Récemment hélas, les performances des caches web ont décrû. De nouveaux protocoles, implémentés comme une couche d'abstraction au-dessus des protocoles web traditionnels comme HTTP, sont devenus extrêmement populaires. Parmi ceux-ci, on trouve les protocoles de messagerie instantanée. Les paquets échangés entre deux machines lors d'une conversation ne présentent évidement aucun intérêt à être répliqués. Il en va de même des échanges *peer-to-peer*, qui ont depuis un certain temps supplanté en nombre les échanges de HTTP « pur » (les paquets échangés pendant la navigation, etc.). Se sont également développés les langages web dynamiques tel que le PHP. En effet, une page écrite en PHP ou dans un autre langage web dynamique renvoie un document HTTP différent en fonction des paramètres en entrée, de la base de données du serveur ou encore des cookies enregistrés sur l'ordinateur de l'utilisateur[1].

Le fonctionnement actuel des caches web, consistant simplement à répliquer les données web échangées n'est plus suffisant. Répliquer une conversation n'augmente absolument pas les performances d'un logiciel de messagerie instantanée. Les caches web nécessitent donc un certain nombre d'évolutions,

[1]Le web dynamique pose des difficultés dans d'autres domaines, comme par exemple l'indexation des pages par les moteurs de recherche

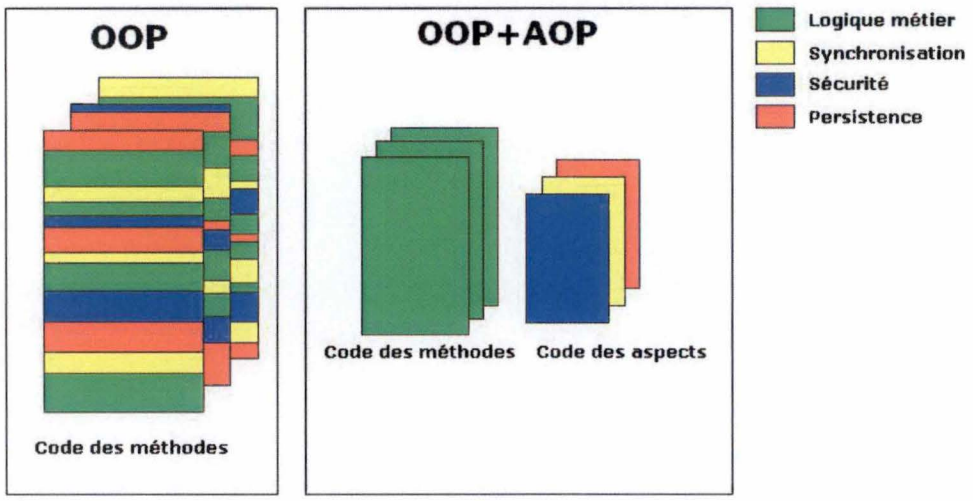


FIG. 3.1 – Dépasser les insuffisances de l'orienté-objet grâce à l'orienté-aspect. Comme on l'observe sur la figure de droite, l'orienté-aspect permet d'augmenter encore la modularité déjà autorisée par l'orienté-objet.
Source : http ://www.dotnetguru.org/articles/dossiers/aop/quid/AOP15.htm

et sous peine de devoir désactiver temporairement l'accès à Internet à un grand nombre d'utilisateurs, ils doivent évoluer dynamiquement.

### 3.2.1 Programmation orientée-aspect

Arachne est un tisseur d'aspects pour le langage C. La programmation orientée-aspect (AOP) est fondée sur l'idée d'une meilleure séparation des préoccupations (*concerns*). L'idée est de dépasser l'abstraction des mécanismes orientés-objet qui, certes, offrent un certain niveau de modularité mais qui demeurent hélas insuffisants pour modulariser toutes les préoccupations [28]. Dans un système complexe, certaines préoccupations sont intrinsèquement transversales, c'est-à-dire que le code qui les implémente est disséminé à travers tous les modules de l'application. La figure 3.1 schématise cette problématique. Les exemples de préoccupations transversales (*cross-cutting concerns*) les plus répandus sont le logging ou certaines exigences non-fonctionnelles telles que la sécurité, la persistance, etc.

L'AOP fournit des mécanismes qui permettent de capturer explicitement ces préoccupations transversales. Ces préoccupations, à l'instar de celles qui étaient modularisées grâce aux apports de l'orienté-objet, peuvent alors être

programmées elle aussi de façon modulaire, c'est-à-dire plus facilement, plus clairement, plus lisiblement. Idéalement, une préoccupation donnée devrait être soigneusement encapsulée dans un objet ou, quand cela n'est pas possible, implémentée par un aspect. Les différents outils AOP partagent une même terminologie :

- **Le tisseur d'aspects** (*aspect weaver*). Le design d'un système, idéalement et grâce aux mécanismes de l'AOP, devrait être semblable à la partie droite de la figure 3.1. Or, à l'exécution, c'est bien un logiciel tel que celui représenté sur la partie gauche qui doit s'exécuter. Le code contenu dans les aspects et le code de base (représenté en vert sur la figure 3.1) doivent être mis en commun, cousus, tissés avant l'exécution. C'est ce processus que l'on nomme tissage ou *weaving*. L'outil qui réalise ce processus est un tisseur d'aspects.

- **Le joinpoint** désigne un point d'entrée potentiel dans le code de base, un point d'où le flot d'exécution peut être détourné vers du code contenu dans un aspect. Dans AspectJ [28], outil AOP dédié au langage Java, les joinpoints peuvent être des déclarations ou des appels de méthodes, de constructeurs ou d'exceptions, des branchements conditionnels, des assignations, etc. Dans le cas du logging par exemple, si nous considérons chaque déclaration de méthode d'une classe, il est possible d'écrire un aspect qui détourne le flot d'exécution au niveau de chaque appel de méthode et qui écrit un message à l'écran avant d'exécuter leur code. Ainsi, durant l'exécution, tout appel d'une méthode de cette classe générera un message à l'écran.

- **Le pointcut** désigne une collection de joinpoints. Il est construit comme une expression qui permet de filtrer les joinpoints qui appartiennent à cette collection. Si un joinpoint correspond à pointcut particulier, le code de l'aspect y correspondant est appelé. Sinon, l'exécution du code de base se poursuit normalement. L'expression suivante est un pointcut AspectJ. Il filtre les méthodes qui renvoient un type quelconque (représenté par une étoile) et dont le nom commence par « set » :

  ```
  pointcut set() : execution(* set*(..) )
  ```

- **Le code advice** est le code contenu dans un aspect. Un code advice est exécuté quand on trouve dans le flot d'exécution un joinpoint qui correspond à un pointcut. Pour reprendre l'exemple précédent, quand on trouve une méthode dont le nom est préfixé par « set », on peut associer le code

advice suivant, qui affiche un message à l'écran après l'exécution de la méthode filtrée par le pointcut :

```
after () : set() {
     System.out.println("Fin d'une méthode préfixé par SET");
}
```

La technologie orientée-aspect répond particulièrement bien aux besoins d'évolution des caches web. Ces systèmes nécessitent en effet des évolutions permettant de prendre en compte les nouveaux protocoles qui sont aujourd'hui largement utilisés sur Internet. Ces évolutions sont, de par la nature de l'architecture et du fonctionnement des caches web, intrinsèquement transversales.

### 3.2.2 Utilisation

Voici un exemple d'aspect Arachne tiré de [36]. La grammaire complète est disponible en annexe :

```
require httpHeaderGetByName as char * (*)(Headers headers, char* name);

void miss(char * uri, HttpHeader header) {
    /* Do many useful things here using httpHeaderGetByName */
    return ;
}

MissTrigger [:
   /* pointcut */
   void clientProcessMiss(clientHttpRequest * http) [:{
      /* BEGIN of the code advice */
      miss(http->uri, http->request->header);
      continue(http);
      /* END of the code advice */
   } :]
:]
```

Le pointcut associé à l'aspect `MissTrigger` indique que le code advice va remplacer la fonction `clientProcessMiss`. Arachne, à l'inverse d'AspectJ, ne permet pas explicitement les modalités telles que `after()`, `before()` ou `instead()`. Néanmoins, l'instruction `continue()` permet d'invoquer le code de la fonction originale. Les trois modalités peuvent donc être émulées : on peut ajouter du code qui s'exécute avant ou/et après le code original, ou le remplacer si l'on n'utilise pas l'instruction `continue()`.

Arachne permet l'utilisation du langage C. Dans l'exemple, `miss()` est une fonction C qui agit sur le record `http` utilisé en paramètre. Le code advice est lui aussi écrit en langage C. L'aspect peut également accéder aux fonctions contenues dans le programme de base. Il suffit de l'importer grâce au mot-clef `require`.

L'effet de l'aspect `MissTrigger` est de faire précéder chaque appel de la fonction `clientProcessMiss` du programme de base par un appel de la fonction `miss` sur le paramètre `http`.

Le langage permettant d'écrire les pointcuts semble assez simple, puisqu'il ne permet de filtrer que les fonctions qui correspondent exactement à une chaîne de caractères donnée. Néanmoins, dans les publications les plus récentes, à savoir [20, 23], le langage de pointcuts semble plus sophistiqué.

Une fois les différents aspects écrits, encore faut-il les tisser au programme de base en exécution. Arachne fournit une boîte à outils (*toolbox*) comprenant : `acc`, `weave` et `deweave`.

Tout d'abord, **`acc`**, est le compilateur d'aspects. Il permet de traduire les fichiers `.ac` contenant des aspects, tel que celui présenté dans l'exemple, en librairies natives DDL.

La commande **`weave`** <**`pid`**> <**`aspect-librairy`**> tisse l'aspect compilé en DLL désigné par `aspect-librairy` avec le programme en exécution désigné par son identifiant processus `pid`; tandis que la commande **`deweave`** <**`pid`**> <**`aspect-librairy`**> permet de détisser le programme et l'aspect.

### 3.2.3 Implémentation

Dans Arachne, chaque joinpoint, c'est-à-dire chaque point du programme de base où le flot d'exécution peut-être détourné vers du code advice (les déclarations de fonctions et les accès aux variables globales) est associé à une **ombre** (*shadow*) qui représente le joinpoint dans le programme compilé en code natif. Au moment du tissage, les ombres sont réécrites et remplacées par une instruction `goto` vers ce qu'on appelle le **crochet** (*hook*). Cette étape est représentée dans le bloc « Base Program » de la figure 3.2.

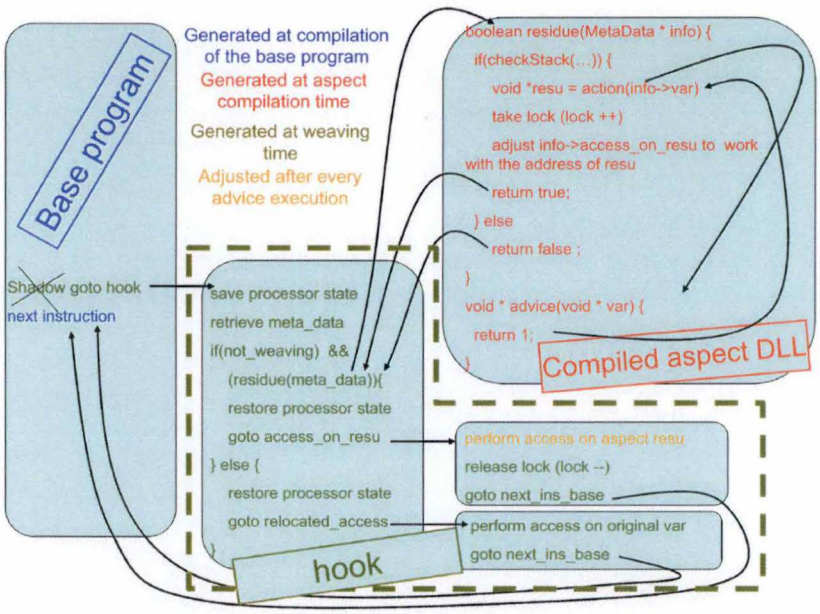


FIG. 3.2 – Au moment du tissage, les ombres (déclarations de fonction ou accès aux variables globales) sont remplacées par un `goto` vers le crochet. Ce dernier détermine (i) si l'aspect est déjà chargé dans le système, (ii) si l'appel de la fonction ou l'accès à la variable globale doit être détourné vers un code advice. Si l'une de ces conditions n'est pas respectée, alors le flot d'exécution reprend au niveau de l'ombre, c'est-à-dire là où il a été détourné dans le programme de base. Source : [36]

On trouve dans [37] le pseudo-code du crochet :

```
<save registers>
if (aspect_loaded && pointcut()) {
   <call aspect>
   <restore registers>
   return <aspect result>
} else {
   <restore registers>
   <perform the original effect of the join point>
   return <result>
}
```

Il s'agit d'un test qui vérifie si le fichier contenant le code de l'aspect est chargé dans l'espace mémoire (condition `aspect_loaded`) et si le joinpoint est associé à un pointcut (condition `pointcut()`). Si c'est le cas, on exécute le code advice associé à ce joinpoint. Sinon, on retourne au programme de base et l'exécution se déroule normalement. Le crochet est représenté sur la figure 3.2 par les points tillés en vert.

### 3.2.4 Discussion

Arachne a été conçu pour répondre à un problème très précis et très localisé. Les évolutions récentes des usages d'Internet ont motivé le besoin d'adaptabilité de Squid, dans lequel aucune évolution ultérieure n'avait été prévue.

L'implémentation liée au code natif limite l'outil aux processeurs Pentium. Généraliser cette solution nécessiterait une nouvelle implémentation pour chaque type de processeur, puisque chaque processeur possède un jeu d'instructions machines qui lui est propre. En outre, l'outil ne fonctionne que sur les systèmes d'exploitation Unix.

Il est également à noter qu'Arachne fait l'hypothèse que le programme de base aura été compilé sans optimisation. Arachne suppose entre autres que le programme de base compilé contient toujours sa table des symboles, afin de pouvoir retrouver les déclarations des fonctions. Néanmoins, les modifications apportées à Squid n'ont pas posé ce genre de problème. D'autres systèmes écrits en C ont été compilé en intégrant des optimisations pour l'exécution. Ceux-ci ne peuvent donc être modifiés dynamiquement avec Arachne.

Pour toutes ces raisons, cette approche ne peut être généralisée et ne répond efficacement qu'à une gamme de problèmes réduite aux systèmes compilés sans optimisation, fonctionnant sous un processeur Pentium et un système d'exploitation Unix.

### Caractéristiques d'analyse

**Granularité :** Arachne permet de détourner le flot d'exécution au niveau des appels de fonctions et des accès aux variables globales (en écriture et en lecture). Il permet d'ajouter des fonctions qui pourront être appelées par les aspects. Il est néanmoins impossible de supprimer une fonction du programme de base.

**Type de modification :** Il n'existe guère dans le langage de C de structure architecturalle forte, si ce n'est l'instruction `require` qui permet d'importer les structures de données et fonctions qui se trouvent dans un fichier. Les modifications autorisées ici sont plutôt de type comportemental, bien que cette introduction de code source puisse entraîner l'insertion de `require` permettant d'importer un large ensemble de structures de données et de fonctions.

**Ouverture :** De par la nature même du problème auquel il répond, Arachne est une solution totalement ouverte, puisque les évolutions ultérieures n'étaient pas prévues lors du design des caches web. Notons néanmoins que la version précédente d'Arachne (appelée $\mu$Dyner) nécessitait d'ajouter le mot-clef `hoockable` devant les fonctions susceptibles d'être détournées. Cette précédente version - purement expérimentale puisqu'elle ne pouvait servir dans le cas de Squid - était quant à elle partiellement ouverte, puisqu'elle nécessitait de prévoir quelles fonctions seraient modifiables en exécution.

**Gestion de l'état courant :** Les appels de fonctions et les accès aux variables globales qui succèdent la phase de transformation et qui sont désignés par des pointcuts sont détournés vers le code advice. Comme nous le verrons dans prochaine section, le comportement est identique en Java, bien que les développeurs de PJama proposent plusieurs stratégies.

## 3.3 PJama

Habituellement, toutes les données créées et manipulées en mémoire lors de l'exécution d'un programme sont automatiquement perdues lorsque celui-ci se termine. Or, de nombreuses applications nécessitent de pouvoir utiliser les objets créés lors des exécutions précédentes. Certains objets doivent survivre aux arrêts de l'application. On appelle **persistance** la préservation des objets (ou par extension, de l'état de l'application) en dehors de la mémoire vive. Les applications persistantes disposent généralement d'un **store** implémenté sur une base de données relationnelle ou orientée-objet.

Les solutions de persistance sont parfois peu pratiques. Elles peuvent par exemple obliger le programmeur à devoir penser lui-même une représentation de l'objet persistant comme un élément de base de données relationnelle. Les nouvelles solutions tendent vers plus de transparence. On parle notamment de **persistance orthogonale** quand une solution repose sur les trois principes suivants :

- **Type Orthogonality** : la persistance est disponible pour toutes les données, indépendement du type. Ceci inclut les objets `Class` et `Thread`[2]..

- **Persistence by Reachability** (qu'on nomme également **persistance transitive**) : la durée de vie des objets est déterminée en fonction d'un ensemble d'objets roots. On élit un ensemble d'objets comme persistants et tous les objets qu'ils utilisent seront automatiquement rendus persistants.

- **Persistence Independence** : aucune modification du code source ou du bytecode n'est nécessaire pour rendre les objets persistants.

PJama est un projet destiné à fournir les moyens de maintenir aisément des applications Java persistantes. En effet, qu'une solution de persistance puisse ou non être qualifiée d'orthogonale, on imagine aisément les difficultés posées par l'évolution des systèmes persistants complexes. Qu'il s'agisse d'évolutions de l'application, de corrections de bugs ou d'autres transformations, il est nécessaire de maintenir la cohérence entre les objets persistants sauvegardés dans le store et les classes que l'on modifie. Sans l'aide des outils idoines, les modifications de l'application nécessiteraient les modifications équivalentes

[2]Bien que la version actuelle de PJama ne permette pas la persistance pour les instances de `Thread`

sur les objets conservés dans le store, sachant que les deux représentations du même objet peuvent être très différentes. Par exemple, si l'on ajoute un attribut à une classe, il faut que les objets sauvegardés disposent aussitôt de ce nouvel attribut. Maintenir cette cohérence manuellement peut être extrêmement laborieux, voire impossible. PJama tend précisément à automatiser le processus d'évolution.

Les problèmes posés par l'évolution des applications persistantes et par l'évolution dynamique sont comparables, puisque dans les deux cas il s'agit de confronter une nouvelle version du logiciel à un ensemble d'objets qui représente l'ancienne version. Tantôt cet ensemble est sauvegardé dans une base de données, tantôt il est chargé en mémoire vive. L'expérience acquise par les développeurs de PJama a pu être exploitée pour l'évolution dynamique. Les procédures de test conçues lors du développement de PJama, destinées à vérifier si une modification donnée est autorisée et ce qu'il convient de faire quand ça n'est pas le cas, peuvent être réutilisées dans le contexte de la reconfiguration dynamique. Par ailleurs, comme dans l'évolution des applications persistantes, les modifications dynamiques sont susceptibles de transformer le format des instances et nécessiter leur conversion. Si l'on ajoute un attribut à une classe par exemple, les instances de cette classe doivent en disposer aussitôt.

L'évolution dynamique demeure néanmoins plus difficile que l'évolution des applications persistantes. Ces dernières sont arrêtées lors de l'évolution et seules sont affectées une collection de classes et leurs instances. Dans le cas dynamique par contre, l'application possède un état supplémentaire, à savoir le contenu de la pile d'exécution des threads.

Puisque l'implémentation de l'évolution dynamique pour Java est un travail de recherche complexe comportant énormément d'aspects techniques et conceptuels qui ne sont pas encore tout à fait maîtrisés, l'implémentation a été décomposée en plusieurs niveaux. Au premier incrément, peu de fonctionnalités ont été introduites, permettant aux développeurs Java peu de possibilités de modifications dynamiques. En contrepartie, la difficulté d'implémentation, qui consistait principalement à devoir modifier le code source de la JVM (écrit en C++), était moindre. Ce premier niveau permet de modifier le corps d'une méthode. Le second niveau permet de procéder à des modifications à un échelon de granularité plus élevé : la classe, dans certaines limites. Nous expliquerons la nature et la portée de ces limites. Le dernier niveau devait permettre de dépasser ces limites et d'autoriser tout type de

modification, mais les développeurs de PJama se heurtent à des difficultés techniques et conceptuelles. Il n'existe guère à ce jour et à notre connaissance d'outil qui permette de dépasser ces limites et d'effectuer n'importe quel type de modification, arbitrairement.

Nous ferons ici abstraction de la majeure partie des détails techniques (transformations des objets en mémoire internes à la JVM, des pointeurs de la table des symboles, de la pile d'exécution, etc.), pour nous concentrer plutôt sur les aspects protocolaires.

### 3.3.1 Niveau 1 : Méthode

Ce niveau de granularité a été choisi en premier par les développeurs de Sun Labs pour plusieurs raisons. Tout d'abord, c'est le niveau de granularité minimum et il était probable que ce soit ce niveau qui nécessite le moins de modifications de la JVM. Ce niveau semblait raisonnablement utile pour procéder à de petites modifications (comme l'ajout de lignes de code permettant de détecter un bug). De plus, ce niveau est supporté par la plupart des debuggers. Il permet en effet de modifier une portion de code et de reprendre l'exécution à partir du code modifié. Ce type de modification est disponible depuis la version de la JDK 1.4 sortie fin 2001.

Comme signalé plus haut, nous faisons ici abstraction de la plupart des détails concernant les modifications des objets internes à la JVM. Il est cependant intéressant de noter que les différentes opérations permettant de transformer une méthode sont effectuées via un *system thread*. Ce type de thread, contrairement aux threads classiques, suspend d'abord l'exécution des autres threads avant de s'exécuter lui-même et désactive le garbage collector. Ces précautions permettent à la JVM de conserver un état cohérent pendant la transition en évitant que des threads concurrents ne viennent interférer avec les éléments en cours de modification. L'application dans son ensemble doit donc être suspendue durant toute l'opération.

Un aspect intéressant relevé par [17] concerne des stratégies pour gérer les méthodes actives, c'est-à-dire les façons dont on peut passer de l'ancienne à la nouvelle version de la méthode modifiée. Plusieurs stratégies sont proposées :

**Stratégie 1 : Transfert « *on-the-fly* »**

Cette première stratégie consiste à identifier un point dans la nouvelle version de la méthode qui corresponde au point d'exécution courant dans

l'ancienne version, et à reprendre l'exécution à partir de ce point. Afin de définir ce point de transition, on compare textuellement le bytecode[3], de la même façon que la fonction `diff` de Unix, afin de détecter des segments de code communs à l'ancienne et à la nouvelle version de la méthode.

Cette technique ne garantit en rien une exécution « correcte » de d'application. Néanmoins, des modifications purement additives et sans effet de bord n'influent pas sur le bon déroulement de l'exécution. Par exemple, ajouter des instructions `println()` pour aider à détecter un bug. La bonne marche de l'exécution est à la responsabilité du programmeur et dépend du type de modification apportée.

En outre, on imagine qu'il peut parfois être difficile de détecter ce point de transition, comme dans le cas, par exemple, où le point d'exécution courant se trouve dans un segment de code supprimé par la modification.

D'un point de vue plus technique, la pile d'exécution empile un *method frame* à chaque invocation. Chaque frame contient les variables locales de la méthode [29]. Il est tout à fait possible que la nouvelle version de la classe nécessite un autre frame (si de nouvelles variables locales ont été déclarées dans la nouvelle version, par exemple) et que les frames des deux versions soient incompatibles, ce qui peut aboutir à des comportements incohérents.

Au vu des problèmes tant conceptuels que techniques posées par cette stratégie, nous concluerons qu'elle ne peut raisonnablement être employée pour réaliser des modifications importantes et durables et n'est véritablement adaptée qu'au debbugage d'une application.

### Stratégie 2 : Attendre la fin des invocations en cours

Cette stratégie est la plus simple à mettre en place et la plus « propre ». C'est notamment la seule dans laquelle deux versions d'une même méthode ne peuvent coexister. L'implémentation de cette stratégie utilise des informations concernant les appels en cours sur chaque méthode. Si l'on désire transformer une méthode donnée, on doit attendre qu'il n'y ait plus aucun appel en cours. Comme dans la stratégie précédente, les threads sont suspendus à cet instant, afin d'éviter qu'un nouveau thread invoque la méthode

[3]En Java, on appelle bytecode les programmes compilés qui sont interprétés par la JVM. Il portent l'extension `.class`

et qu'on doive encore patienter. L'on peut alors procéder au changement de méthode.

Cette stratégie, bien que présentée comme la plus « propre », a quelques inconvénients. Tout d'abord, elle nécessite d'attendre un moment où la méthode ne sera plus exécutée, ce qui peut être long - voire infini - dans un système multi-threading. De plus, rien ne garantit qu'un appel donné se termine. Les boucles de contrôle, du type `while(true)`, ne se terminent jamais, ainsi que très souvent, la méthode `main`.

**Stratégie 3 : Les nouveaux threads seuls appellent la nouvelle version**

Une nouvelle version de la méthode est créée à chaque transformation. Les threads créés avant la transformation appellent l'ancienne version de la méthode, alors que les threads créés après la transformation appellent la nouvelle version. Cette stratégie semble la plus difficile des trois à implémenter puisqu'elle permet à deux versions (ou plus) d'une même méthode de coexister indéfiniment, et ce même en l'absence de boucle infinie. Les threads les plus anciens peuvent ne jamais se terminer et appeler les anciennes versions des méthodes encore et encore.

Plutôt que de procéder à de lourdes modifications de la JVM, pour un objectif finalement limité, l'implémentation pourrait être basée sur la création d'« aiguillages » chargés de décider, pour un thread donné, quelle version il doit appeler. Pour chaque méthode transformée, on conserve l'ancienne version, la nouvelle et une méthode de dispatching dans une classe temporaire. Les appels sont redirigés vers ce dispatcher. Une fois les anciens threads terminés, on peut supprimer définitivement l'ancienne version de la méthode et réorienter les appels du dispatcher à la nouvelle version, afin d'éviter la consommation supplémentaire de ressources qu'entraîne le dispatching. Encore une fois, il est envisageable qu'une situation où les plus anciens threads se terminent n'arrive jamais. Cette stratégie semble néanmoins la plus adaptée dans le cas d'un serveur qui fonctionne en mode *thread-per-request*.

Cette stratégie est complexe à implémenter. Dans le cas où plus de deux versions de la méthode existent, il se peut qu'un thread passe par plusieurs dispatchers imbriqués en fonction du moment où il a été créé.

**Stratégie 4 :**

Il apparaît qu'aucune de ces stratégies n'est universelle. Selon le type de système (*single-threaded*, *thread-per-request*, *thread-per-session*, etc.), selon l'architecture, selon la méthode visée et le type de modification envisagée, l'une ou l'autre stratégie nous paraîtra plus adaptée.

En réalité, aucune des stratégies précédemment décrites n'est effectivement disponible. La stratégie choisie est celle qui a semblé techniquement la plus simple : on laisse les appels en cours sur l'ancienne version se terminer et tous les nouveaux appels ont lieu sur la nouvelle version. Cette stratégie ressemble à s'y méprendre à la solution précédente, si ce n'est qu'elle se base sur une division entre appels plutôt que sur une division entre threads. Encore une fois, il est possible que plusieurs versions d'une même méthode coexistent dans le système si des appels ne se terminent pas.

### 3.3.2 Niveau 2 : Classe

Ce sont les modifications au niveau de la classe qui ont été choisies pour la seconde étape. Les transformations autorisées se limitent aux transformations dites *binary compatibles* [24], qui sont :

- Modifier l'implémentation des méthodes et des constructeurs (ce qui correspond au niveau 1)
- Ajouter de nouveaux champs, méthodes et constructeurs à une classe ou à une interface
- Supprimer un champ, une méthode ou un constructeur `private` d'une classe ou d'une interface
- Changer l'ordre des déclarations des champs, méthodes et constructeurs ou des interfaces
- Déplacer une méthode vers la superclasse
- Insérer de nouvelles classes ou interfaces dans une hiérarchie
- Changer des méthodes ou des constructeurs pour qu'ils retournent des valeurs sur des inputs pour lesquels ils généraient précédemment des exceptions ou des deadlocks

En d'autres mots, il s'agit de modifications qui garantissent que les invariants internes à JVM seront toujours respectés, par exemple ceux qui postulent que toute classe marquée `final` ne peut avoir de sous-classes ou que toute classe marquée `abstract` ne peut être instanciée. Permettre des changements arbitraires imposerait de devoir vérifier une pléthore de contraintes

additionnelles sur l'état de l'application (et plus seulement sur la classe elle-même). Par exemple, nous devrions refuser une modification dans laquelle une classe qui devient `final` a déjà des sous-classes chargées dans le système ou dans laquelle une classe devient `absract` alors qu'elle possède des instances en exécution.

Il s'agit donc de modifications généralement additives et qui ne peuvent « briser » un lien entre deux classes. Supprimer une méthode `public` d'une classe est une transformation *binary incompabible*, puisqu'elle est susceptible de générer des erreurs pour les classes qui appellaient cette méthode. En revanche, supprimer un élément `private` d'une classe peut générer des erreurs à l'intérieur de cette classe mais n'a pas d'influence sur les liaisons entre classes. Supprimer un élément `private` est donc bien *binary compatible.*

Une telle modification se déroule en trois étapes principales :

1. Chargement de la nouvelle version de la classe
2. Validation des changements et choix de la technique de remplacement
3. Réenregistement des nouvelles versions

**1. Chargement de la nouvelle version de la classe**

Une transformation à ce niveau peut entraîner la création d'une nouvelle version de la classe. Rien ne nous empêche d'y ajouter des méthodes et des attributs. Outre le fait que la nouvelle version de la classe peut conduire à devoir modifier l'allocation en mémoire (les objets qui représentent les tables des méthodes et des champs sont inclus dans les objets qui représentent les classes), la possibilité d'effectuer des modifications *binary compatibles* comprend la possibilité d'insérer des classes dans une hiérarchie existante, comme illustré sur la figure 3.3.

Conséquement, à l'inverse du niveau précédent, l'objet C++ représentant l'ancienne version de la classe en mémoire ne peut plus être utilisé pour représenter la nouvelle version. D'autre part, le fait de pouvoir modifier la hiérarchie entre classes (cf. figure 3.3) modifie les liens entre les classes déjà chargées. Pour toutes ces raisons, le `ClassLoader` a été réimplémenté pour devenir le `HotSwapClassLoader`. Celui-ci possède, par rapport au `ClassLoader` classique, une méthode supplémentaire

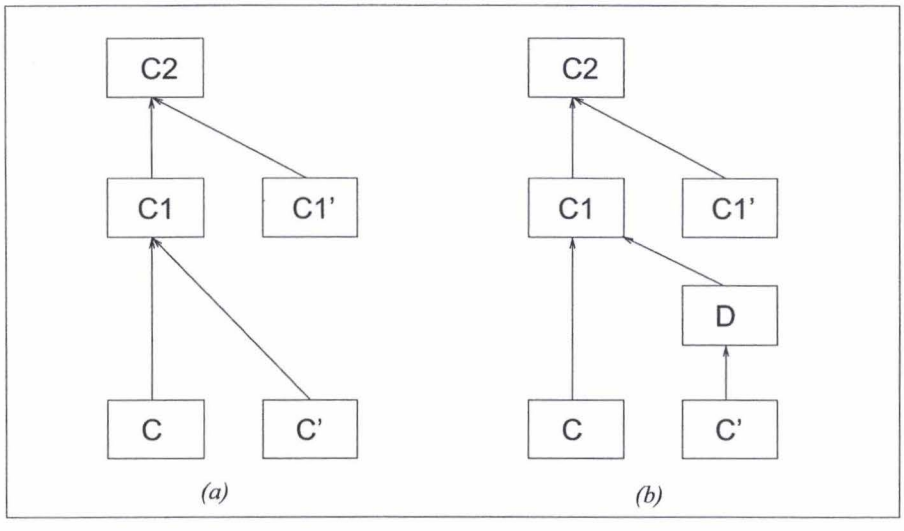


FIG. 3.3 – Insérer de nouvelles classes dans une hiérarchie existante. On peut modifier la déclaration `extends` de la classe `C'`, de telle façon que celle-ci étende à présent `D`. Dans ce cas, il faut que `D` étende `C1` directement ou indirectement. En effet, si dans l'ancienne version de l'application, on invoque des méthodes de `C1` sur la classe `C'`, il faut que dans la nouvelle version, `C'` soit encore en mesure de fournir les méthodes de `C1`. Source : [17]

`defineStartingClass(Class C)` qui est appelée par la JVM pour notifier le `HotSwapClassLoader` qu'une nouvelle version de la classe `C` (passée en paramètre) va être chargée. De cette façon, la `HotSwapClassLoader` peut retrouver la référence du class loader original attaché à `C` afin de localiser et de charger ses superclasses qui ne sont pas affectées par la modification.

### 2. Validation des changements et choix de la technique de remplacement

Une fois les nouvelles versions des classes transformées et les nouvelles classes chargées dans le système, il est nécessaire de comparer chaque couple ancienne-nouvelle version de classe afin de valider les transformations et de s'assurer qu'elles sont bien *binary compatibles*, et d'annuler toute l'opération si ça n'est pas le cas.

Cette étape détermine également la technique de remplacement : si les transformations sont limitées au corps des méthodes, alors la technique est identique à celle décrite plus haut. Sinon, d'autres opérations seront nécessaires.

On vérifie d'abord les modifieurs, c'est-à-dire ces mots-clés qui permettent des contraintes de programmation, que nous avons déjà brièvement abordés.

- Les modifieurs d'accès (`private`, `public` et `protected`) ne peuvent être plus contraignants dans la nouvelle version, au risque de briser les invariants de la JVM. Ainsi, les méthodes et attributs déclarés `private` dans l'ancienne version peuvent devenir `public` dans la nouvelle, mais non l'inverse. En effet, imaginons qu'une classe `A` utilise la méthode `public m()` de la classe `B`. Si cette méthode devient `private`, alors `A` n'y a plus accès.

- Vérifier que les classes et les méthodes `final` ou[4] `abstract` demeurent respectivement `final` ou `abstract`.

On vérifie également que la superclasse de l'ancienne version est toujours l'une des superclasses (directe ou indirecte) de la nouvelle version. Par exemple, si la classe `C1` de la figure 3.3 possède une méthode `m()`, la classe `C'` la possède également, par héritage. Si une nouvelle classe `D` est insérée dans la hiérarchie, il est nécessaire que `C'` possède toujours `m()` puisque celle-ci peut être appellée sur `C'`. Il est donc nécessaire que `C1` demeure l'une des superclasses - directe ou indirecte - de `C'`.

Enfin, on vérifie pour chaque classe que la nouvelle version implémente directement ou indirectement les interfaces qu'implémentait directement l'ancienne version.

**3. Réenregistement des nouvelles versions**

Cette étape doit également avoir lieu dans *system thread* et le garbage collector doit être temporairement désactivé. Elle consiste essentiellement à réorganiser les allocations en mémoire de l'application. Par exemple, les classes auxquelles on a ajouté des méthodes et des champs doivent généralement être relocalisées.

### 3.3.3 Discussion

Le modèle de programmation de Java est très riche. Il permet l'héritage, le polymorphisme, la protection des modifieurs d'accès, etc. Plus précisément, on dit qu'il existe une référence symbolique résolue de C1 à C2 (où C1 et C2

[4]Les deux à la fois étant logiquement impossible.

sont des classes ou des interfaces) si l'une de ces conditions est remplie [24] :

- C2 est superclasse de C1
- C1 implémente C2
- C2 est utilisé comme type de variable , comme type de paramètre, comme type de retour d'une méthode ou d'un constructeur de C1
- C1 utilise une variable statique de C2

Cette solution est plus abstraite qu'Arachne mais demeure proche du code source. Le modèle de programmation impose aux opérations de reconfiguration dynamique de vérifier énormément de contraintes pour demeurer conforme au modèle et à la spécification du langage. C'est précisément la complexité du modèle de programmation qui limite la portée des modifications.

Le problème de la gestion de l'état courant est particulièrement développé dans [19, 18]. Outre les différentes stratégies proposées pour passer de l'ancienne à la nouvelle version d'une méthode, le développement préalable de PJama a poussé les développeurs à penser le problème de conversion des instances.

### Conversion des instances

Lorsque l'on instancie une classe, on crée un objet à l'image de cette classe. Il existe donc une relation particulière entre une classe et ses instances. Si l'on transforme la classe, il se peut que l'on affecte cette relation et qu'on doive donc transformer les instances afin de préserver cette bijection. Si l'on ajoute des attributs ou des méthodes à une classe, il faut que les objets en disposent également, au risque de briser ce lien ou de créer plusieurs versions d'objet pour une même classe.

Cette conversion peut être immédiate (*eager conversion*) ou paresseuse (*lazy conversion*). La conversion immédiate implique que tous les objets soient convertis en une fois dès que la classe est modifiée. La conversion paresseuse par contre, retarde la conversion de chaque objet au moment où il est accédé. Des politiques intermédiaires sont envisageables. On peut diviser les objets en groupes et convertir le groupe entier lorsqu'un des objets du groupe est accédé.

On peut également imaginer une stratégie de conversion par défaut (*default conversion*) ou personnalisée (*custom conversion*). Dans une technique par défaut, les données sont copiées automatiquement. Les attributs qui n'ont pas changé gardent leur valeur, les nouveaux attributs sont initialisés avec une valeur par défaut et ceux qui sont supprimés perdent leur valeur. Dans le cas d'une conversion personnalisée, on offre au programmeur la possibilité d'écrire du code de conversion plus spécifique. Par exemple, si un champ `String nom` est scindé en `String nomDeFamille` et `String prenom`. Ou encore, si l'on change la représentation d'un point, en passant de la représentation cartésienne à la représentation polaire. Dans les deux cas, on peut laisser au programmeur la possibilité d'écrire du code pour effectuer la conversion.

L'implémentation actuelle de PJama permet, dans le cas des systèmes persistants, la conversion immédiate par défaut et dans une certaine mesure, la conversion immédiate personnalisée. La conversion paresseuse pourrait être intéressante dans le cas de l'évolution dynamique puisqu'elle permettrait d'améliorer les performances. Toutefois, si la nouvelle version des objets est plus volumineuse, il se peut que l'objet doive être relocalisé dans l'espace mémoire, sachant que les pointeurs entre les objets sont directs.

En effet, dans les versions plus anciennes de la JVM, une référence d'un objet vers un autre était en réalité un pointeur vers une zone de la pile, qui pointait à son tour vers l'emplacement effectif de l'objet référencé. Relocaliser un objet nécessitait de changer une seule référence, à savoir l'adresse sur la pile. Dans une telle organisation, l'accès à un objet référencé passait par deux pointeurs successifs. Pour des raisons de performances, on est passé à une organisation où les références entre objets sont directes. Ceci implique alors que si on relocalise un objet, l'on doive changer toutes les références qui pointent vers lui, et non plus seulement l'adresse sur la pile.

### Les transformations *binary incompatibles*

La solution présentée impose beaucoup de restrictions. Les transformations *binary incompatibles* en effet posent un certain nombre de problèmes techniques et conceptuels : elle peuvent potentiellement rompre des invariants internes à la JVM, telle que la règle qui postule qu'une classe marquée `final` ne peut avoir de sous-classes. Ces invariants sont normalement vérifiés par la JVM lors du chargement et de la liaison des classes (voir JVMS 2.17.2 et 2.17.3 [29] pour plus d'informations), et ces tests auront lieu sur chaque nouvelle classe chargée par la JVM. Or, une classe que l'on désire

transformer est déjà chargée dans la JVM, d'autres classes l'utilisent probablement et des liens ont déjà été établis avec elle : on utilise ses variables statiques, elle est étendue, implémentée, utilisée comme type de variable, de paramètre ou de retour. Les modifications *binary incompabiles*, par définition, sont susceptibles d'invalider ces liens.

Précisons que ces transformations sont *susceptibles* d'induire des problèmes de liaisons, mais ça n'est pas forcément le cas, d'où l'intérêt d'une implémentation qui les permettraient malgré les difficultés posées. Imaginons deux classes `A` et `B`. Si une méthode publique de `B` n'est appelée que par une méthode privée de `A` et par elle seule, supprimer ces deux méthodes est bien *binary incompatible* mais n'induit pas d'erreur.

Pour implémenter un outil qui permette effectivement ce type de transformation, nous devrions vérifier qu'aucune contrainte n'est violée et annuler toute l'opération dans le cas contraire. L'option la plus sage semble être de répéter les tests que l'on exécute normalement lors de la créations des liaisons pour toutes les classes utilisées par la classe que l'on modifie, afin de s'assurer qu'aucune de ces transformations *binary incompatibles* n'induise effectivement de problème de liaison.

Hélas, les choses ne sont pas aussi simples. D'un côté, on pourrait écrire du code de vérification semblable au code existant déjà dans la JVM. D'un autre côté, si l'on tente de réutiliser ce code tel quel, il est très probable qu'on rencontre un certain nombre de problèmes dus aux divergences de représentation des classes lors d'une utilisation normale de ce code et lors de l'utilisation pour des modifications à la volée. Par exemple, lors de l'exécution, le bytecode a subi un certain nombre d'optimisations et le code de vérification peut avoir besoin du bytecode original. En plus, il faut éviter que les objets utilisant la classe redéfinie ne pointent sur elle avant que les transformations aient été validées, alors que le code de vérification se base sur l'hypothèse que ces pointeurs sont déjà en place. Il semble donc que l'implémentation d'un outil supportant les modifications *binary incompatibles* implique une inspection rigoureuse du code de vérification interne à la JVM et sa réimplémentation pour le contexte de l'évolution dynamique, ce qui serait sans aucun doute un travail de titans.

Notons qu'Arachne fait également l'hypothèse que le programme de base ait été compilé sans avoir subi certaines optimisations. Comme Arachne, les évolutions dynamiques *binary incompabiles* seraient limitées à du code

compilé sans optimisation, qui contiendraient toutes les méta-informations nécessaires à l'évolution, telle que la table des symboles.

Un autre problème concerne le moment auquel on doit signaler une erreur. Selon la JVMS [29], si une erreur survient, une exception doit être lancée la première fois où le programme utilise le référence qui pose problème. Par exemple, si une classe `C` contient l'instruction `new` alors qu'elle est marquée `abstract`, le programme fonctionnera sans problème jusqu'à ce que cette instruction soit exécutée. Mais que doit-on faire si la classe `C` précédement non-abstraite devient abstraite suite à des opérations de reconfiguration, si l'opérateur `new` a déjà été exécuter plusieurs fois sans problème ? Ceci dépasse le cadre de la JVMS. L'option la plus sage consistant à reporter le problème au plus tôt impliquerait également de réutiliser du code de vérification pour l'adapter au contexte particulier des modifications à chaud.

### Caractéristiques d'analyse

**Granularité :** Les modifications ont lieu sur la classe. Le cas d'une modification d'une méthode est inclu dans le cas de la classe, puisque réimplémenter une méthode est *binary compabile*. Néanmoins, toutes les transformations sur une classe ne sont pas autorisées, puisqu'elles sont limitées aux modifications *binary compatible*. Précisons également que le fonctionnement dynamique de la JVM, et plus précisément du *class loader*, autorise le chargement des classes à chaud. De nouvelles classes peuvent donc être ajoutée à la volée.

**Type de modification :** Remarquons tout d'abord que le concept de classe mélange les aspects structuraux (cf. la référence symbolique) par les déclarations `extend` ou le simple fait pour une classe de posséder un champ dont le type est une autre classe, etc. et les aspects comportementaux, puisqu'elle contient l'implémentation de ses méthodes.

Le premier niveau se limite aux modifications des méthodes. En changeant l'implémentation d'une méthode, on travail à un niveau comportemental. Au second niveau, on permet des modification au niveau de la classe. On peut donc transformer les liens qu'entretiennent les classes entres elles et donc, modifier la structure de l'application.

**Ouverture :** Comme dans la solution précédente, les besoins d'évolution dynamiques ont été pensés *a posteriori*. La solution est donc totalement ouverte puisqu'elle est applicable à tous les applications déjà en exécution.

Une nuance s'impose néanmoins. L'implémentation de la solution a nécessité d'importantes transformations de la JVM. Les applications exécutées sur une version antérieure de la JVM ne pourront donc pas être modifiées dynamiquement. Dans ces cas, il serait préalablement nécessaire d'arrêter l'application le temps de la redéployer sur une version plus récente de la JVM.

En outre, démarrer la JVM dans un mode tel qu'il permette d'effectuer ces modifications nécessitent l'introduction d'un paramètre VM.

**Gestion de l'état :** L'apport de cette solution en ce qui concerne ce critère est l'un des plus complets. D'une part, l'outil propose un mécanisme de conversion des instances personnalisé, qui laisse une assez large marge de manoeuvre au programmeur, qui a la possibilité d'écrire du code Java qui décrive précisément le transfert d'état. D'autre part, bien qu'une seule stratégie soit effectivement disponible (l'arbitrage par appel, ce qui est finalement identique au comportement d'Arachne), les développeurs de PJama proposent d'autres stratégies qui pourraient s'avérer plus efficaces dans certaines situations (le transfert à la volée, l'attente jusqu'à la fin des appels en cours ou l'arbitrage par date de création des threads).

## 3.4 Fractal

Les deux solutions précédemment étudiées nous montrent les difficultés lorsque l'on pense la reconfiguration à un niveau d'abstraction très bas, c'est-à-dire au niveau du code source, voire au niveau du code natif dans le cas d'Arachne. Des difficultés tant techniques (liées à l'implémentation) que conceptuelles (liées au modèle de programmation) limitent la portée des transformations dynamiques.

Dans les deux solutions précédentes, les appels qui précèdent l'opération de reconfiguration ont lieu sur l'ancienne version de la fonction et les appels qui succèdent à l'opération ont lieu sur la nouvelle version. Puisque rien ne garantit qu'un appel donné se termine (boucles de contrôle, serveurs, etc.), la coexistence *ad vitam æternam* d'au moins deux versions de la fonction est possible. En Java plus précisément, chaque nouvel appel a lieu sur la dernière version de la méthode (voir la stratégie 4 décrite à la page 30). Si aucun de ces appels ne termine, les différentes versions sont condamnées à cohabiter.

Par ailleurs, les deux solutions sont purement additives. Avec Arachne, on peut ajouter des fonctions, réimplémenter une fonction ou ajouter du code avant ou après l'appel, mais pas la supprimer. En Java, à cause des limites du *binary compatible* au-delà desquelles peuvent se romprent les références qui lient entre eux les objets, on ne peut supprimer d'élément déclaré `public`. Si dans les deux cas, on a la possibilité de remplacer le corps d'une fonction ou d'une méthode par du code vide, la présence inopportune des en-têtes obsolètes rendrait l'évolution ultérieure plus pénible. En outre, ces fonctions vides pourraient toujours être appelées sans qu'on le remarque et entraîner une consommation supplémentaire de ressources non négligeable.

Rappelons que dans les logiciels de vol embarqués dans les satellites, il est parfois impératif de supprimer du code pour des raisons de sécurité, mais aussi parce que les ressources en EEPROM sont très limitées. L'accumulation de plusieurs versions d'une même portion de code dans le domaine des logiciels de vol est particulièrement problématique.

Il est également à noter que pour ces deux solutions, les mécanismes de reconfiguration ont été introduits après coup et les langages n'avaient pas été initialement prévus pour être reconfigurables. En C comme en Java, les modifications ont lieu à un « méta-niveau » où sont représentés les objets

en exécution et où ces représentations ont été optimisées pour l'exécution et pour aucune « méta-opération », telle qu'une reconfiguration dynamique.

Pour toutes ces raisons, notre attention va maintenant se porter sur un autre paradigme architectural : l'orienté-composant. Le composant est définit par Szyperski [40] comme « une unité de composition qui possède des interfaces spécifiées contractuellement. Un composant logiciel peut être déployé indépendamment et peut être composé avec des composants tiers ».

De tels modèles sont basés sur une division de l'état entre composants. Ils permettront d'introduire des mécanismes *a priori* permettant de modifier l'application, éventuellement en garantissant l'existence de « points d'entrée » dans le cas des solutions partiellement ouvertes.

Fractal est un modèle de composants extensible et modulaire indépendant d'un langage de programmation particulier et permettant de construire l'architecture, d'implémenter et de déployer des applications complexes. Le but de Fractal est de réduire les coûts de développement, de déploiement et de maintenance. Le modèle tend à être applicable à une large gamme d'applications : systèmes embarqués, système d'exploitation, serveurs, système d'informations, etc.

Il ne s'agit guère d'une spécification rigide que devraient respecter les composants mais plutôt d'un système extensible de relations entre des concepts bien définis et d'un ensemble d'APIs que chaque composant peut implémenter, selon les besoins des modélisateurs. Certaines de ces APIs définissent des opérations permettant à un composant d'être observé (capacités d'introspection ou d'observabilité) ou transformé (capacités de reconfiguration). D'autres APIs ont des buts plus fonctionnels, telles que la gestion d'une liste d'attributs ou du cycle de vie.

Ainsi, il est tout aussi possible d'utiliser le modèle de Fractal pour concevoir une application totalement rigide où aucun des composants n'implémenterait les opérations de reconfiguration ; qu'une application très flexible, entièrement modulable. Entre les deux, on peut imaginer des applications dont on déciderait d'une base architecturale fixe et dont l'autre partie serait modulable.

Non seulement les applications modélisées en Fractal peuvent évoluer, mais le modèle de composants lui-même tend à être extensible et adaptable aux

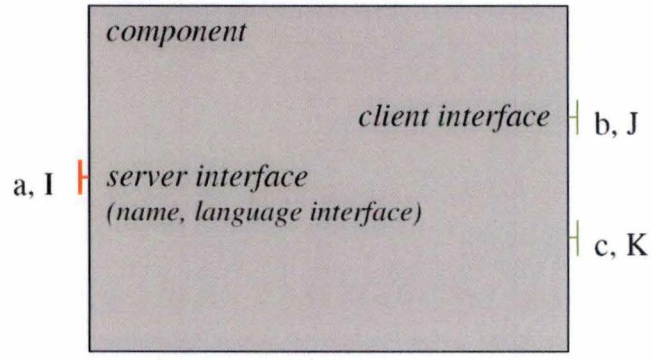


FIG. 3.4 – Vue externe d'un composant Fractal. Les interfaces clients sont représentées en vert et les interfaces serveurs en rouge. Source : [4]

besoins d'un domaine particulier. Le modèle Fractal est volontairement réduit à l'extrême, fournit une sémantique minimaliste dans ce but d'évolutivité, évitant par là même certaines difficultés conceptuelles inhérentes aux modèles plus sophistiqués.

### 3.4.1 Le modèle de composants Fractal

Le modèle se décline en deux vues principales. Dans la première, le composant est vu de l'extérieur (figure 3.4), selon une perspective *black box*, qui donne accès à ses interfaces externes. Comme dans la plupart des modèles orientés-composant, les interfaces sont de deux types : les interfaces clients (qu'on dit parfois *require*, *receptacles* ou *source*, en vert sur les figures) émettent des invocations tandis que les interfaces serveurs (qu'on dit parfois *provided* ou *sink*, en rouge sur les figures) les reçoivent. Les interfaces clients demandent des services et les interfaces serveurs y répondent.

Fractal prévoit à ce niveau des mécanismes d'introspection. L'introspection est décrite par [3] comme la « capacité d'un programme à examiner sa propre structure ». Si l'on considère une opération de reconfiguration dynamique comme une « écriture » sur le système, l'introspection (ou l'observabilité) correspond à une opération de « lecture ». L'introspection permet d'obtenir des méta-informations sur une application.

Notons au passage que le langage Java possède également des capacités d'introspection. Le package `java.lang.reflect` permet d'obtenir de l'information à propos des classes et des objets. Par exemple, le programme de

```
import java.lang.reflect.Method;

public class Example {
    public static void main(String[] args) {
        Method[] m = Object.class.getMethods();
        for (int i = 0; i < m.length; i++) {
            System.out.println(m[i]);
        }
    }
}
```

FIG. 3.5 – Capacités introspectives en Java. Le package `java.lang.reflect` permet d'obtenir des informations à propos des classes et des objets. Il comprend entre autres les classes `Method`, `Constructor`, `Field`, etc.

la figure 3.5 imprime la liste des méthodes (nom, visibilité, modifieurs, arguments, exceptions, etc.) de la classe `Object`.

Dans la vue externe du modèle Fractal, l'introspection se caractérise entre autres par la possibilité de demander à un composant quelles sont ses interfaces : sur chaque composant on peut invoquer `getFcInterfaces()` qui renvoie la liste des interfaces[5]. Sur chaque interface, on peut également invoquer des opérations introspectives et obtenir des informations telles que le type, le nom ou le composant dont elle est issue.

L'autre vue (figure 3.6), interne, se décompose à son tour en deux niveaux. D'abord le **contrôleur** (*controller* en anglais, également appelé membrane) et le **contenu** (*content*). Le contenu possède récursivement d'autres composants, qu'on n'appelle sous-composants (*sub components*). Le contrôleur peut avoir des interfaces internes (accessibles par les sous-composants) ou externes (accessibles de l'extérieur). Une liaison (*binding*) représente un lien de communication entre deux interfaces : l'une client et l'autre serveur. Cette dernière doit accepter au moins toutes les opérations que l'interface client peut émettre. En d'autres mots, le type de l'interface serveur doit être un sous-type ou du même type que l'interface client. Une interface client ne peut être liée qu'à une seule interface serveur alors qu'une interface serveur peut être liée à plusieurs interfaces clients.

Un composant peut implémenter un certain nombre de contrôleurs, accessibles via une interface externe. Sur les schémas Fractal, on représente leur

[5]L'API détaillée de Fractal se trouve dans l'annexe

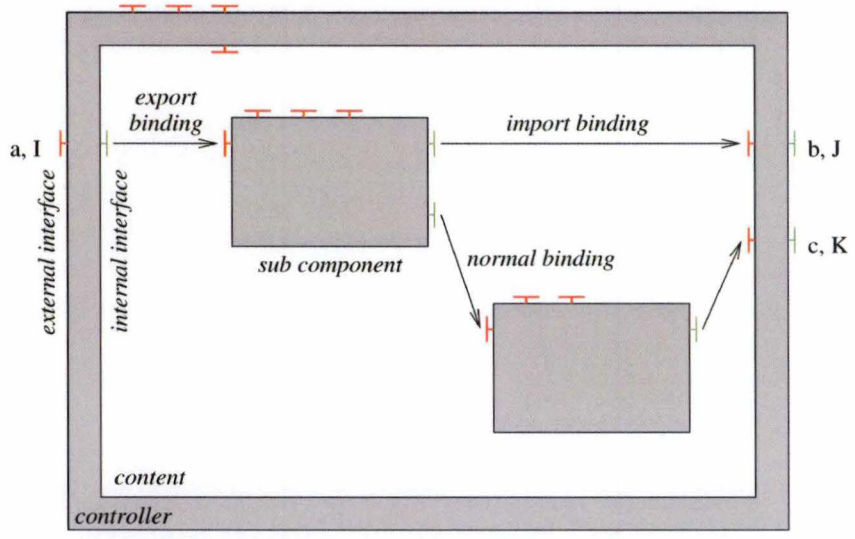


FIG. 3.6 – Vue interne d'un composant Fractal. Un composant peut avoir des interfaces internes ou externes. Source : [4]

interface sur le côté supérieur du composant, à gauche.

### Attribute Controller

Un composant peut posséder des attributs, ce qu'on appelle parfois des champs. Ils reflètent des propriétés relatives au composant auquel ils appartiennent. Ils peuvent être écrits ou lus depuis l'extérieur du composant grâce à l'interface externe `AttributeController`. En réalité, cette interface ne contient aucune opération : elle doit être sous-typée et augmentée des getters et/ou setters que l'on souhaite.

### Binding Controller

Ce contrôleur sert à lier et à délier des interfaces clients et serveurs entre elles à l'intérieur d'un composant. Il contient également des opérations d'introspection : l'opération `lookupFc(string clientItfName)` renvoie l'interface serveur liée a l'interface client dont le nom est passé en paramètre. L'interface `BindingController` contient également les opérations `bindFc(string clientIftName, any serverItf)` et `unbindFc(string clientIftName)` qui permettent respectivement de lier et de délier des interfaces.

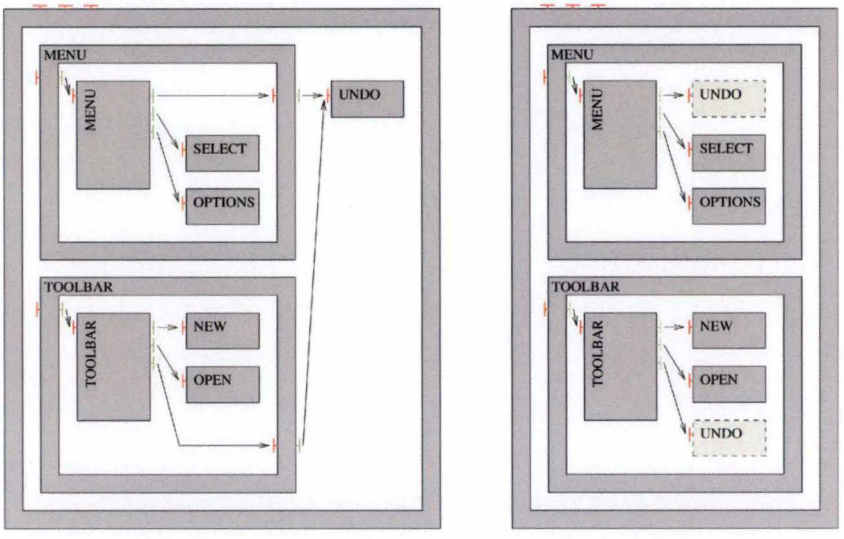


FIG. 3.7 – Composants partagés. A gauche, une application sans possibilité de partage de composants. A droite, la même application où l'on permet de partager un sous-composant. Source : [4]

### Content Controller

D'un point de vue introspectif, ce contrôleur permet d'obtenir la liste des sous-composants (opération `getFcSubComponents()`) et des interfaces internes (opération `getFcInternalInterfaces()`). Le contrôleur fournit également des opérations de reconfiguration : `addFcSubComponent(Component c)` pour ajouter un sous-composant et `removeFcSubComponent(Component c)` pour en enlever. A la différence des solutions précédentes, il existe des mécanismes explicites qui permettent de supprimer des éléments .

Un sous-composant peut être ajouté à plusieurs composants parents. Un tel sous-composant est dit partagé (*shared component*). Ce concept est fort utile - paradoxalement - pour augmenter l'encapsulation. Comme l'illustre la partie gauche de la figure 3.7, sans composant partagé, le sous-composant `UNDO` est accessible non seulement depuis l'intérieur des composants `MENU` et `TOOLBAR`, mais également en dehors de ceux-ci. Tandis qu'avec les composants partagés, comme sur la partie droite de la figure 3.7, `UNDO` n'est accessible qu'à l'intérieur des composants `MENU` et `TOOLBAR`. Les composants partagés peuvent aussi dans une certaine mesure émuler des comportements du type orienté-aspect. L'on peut par exemple trouver un sous-composant (le même) de logging dans plusieurs composants, et le retirer lorsque l'on n'en a

plus besoin. La structure de Fractal en terme de sous-composants directs et indirects se décrit plutôt comme un graphe acyclique que comme un arbre.

### Life Cycle Controller

Comme nous l'avons déjà souligné pour Java, les transformations dynamiques, si elles sont effectuées sans prendre certaines précautions, peuvent entraîner des erreurs ou aboutir à des états incohérents. De même dans Fractal, transformer un attribut, changer une liaison en exécution, peut être dangereux. Des messages peuvent être perdus, l'application peut crasher ou son état peut devenir inconsistant.

Le contrôleur du cycle de vie permet d'aider à assurer la consistance de l'application lors de reconfigurations dynamiques. L'interface fournit deux opérations : `startFc()` et `stopFc()` pour démarrer et arrêter un composant correctement. Pour plus de généricité, la sémantique du modèle est réduite à sa plus simple expression. Ainsi, ces opérations pourraient être récursives (c'est-à-dire que l'arrêt d'un composant pourrait entraîner l'arrêt de ses sous-composants directs et indirects), mais cela n'est pas imposé par le modèle. Le modèle ne précise pas non plus l'effet de ces opérations sur l'état interne. L'opération `stopFc()` pourrait être invoquée pour arrêter proprement le composant avant de le supprimer et son l'état interne serait perdu. Mais l'opération pourrait simplement suspendre l'exécution du composant et son état serait dans ce cas conservé.

L'interface fournit également une opération introspective `getFcState()` qui renvoie l'état du contrôleur : `STARTED` ou `STOPPED`.

Dans l'état `STARTED`, qui peut implicitement signifier que les sous-composants sont dans le même état, le composant peut accepter ou émettre des invocations. On peut imaginer une implémentation qui génère l'erreur `IllegalLifeCycleException` lorsqu'une opération comme `unbindFc()` ou `removeFcSubComponent()` est invoquée. Encore une fois, cela dépend de l'implémentation du modèle.

Dans l'état `STOPPED`, un composant ne peut émettre d'invocations et ne peut en recevoir qu'au niveau des interfaces de contrôle. Le résultat d'une invocation sur une interface fonctionnelle (c'est-à-dire les interfaces écrites par les modélisateurs, ce qui exclut celles des contrôleurs) d'un composant arrêté n'est pas défini. Elle peut aboutir à un résultat normal, renvoyer une

exception, être postposée jusqu'au réveil du composant ou - pourquoi pas - provoquer le redémarrage du composant. Le modèle ne dit pas non plus ce qu'il se passe si le composant est arrêté alors que des invocations ont toujours lieu. Sont-elles annulées et retardées jusqu'au redémarrage du composant ? Peut-on les laisser terminer ? Dans ce cas, il existe toujours le risque que l'appel ne se termine jamais, bien que ce risque soit moindre dans une architecture orientée-composant que dans une architecture orientée-objet ; car les architectures orientées-composant tendent précisément vers un moindre degré de couplage. Les flots d'exécution y sont par nature moins transversaux. Dit autrement, si les boucles sans fin ne sont pas forcément synonymes d'erreur, il est plus sage de les placer au sein même des composants, que l'on peut forcer à s'arrêter.

D'autres composants peuvent nécessiter des cycles de vie différents. Le contrôleur pourrait éventuellement être étendu et adapté pour inclure des états plus sophistiqués. Le modèle exige néanmoins que soit conservée la sémantique des états `STARTED` et `STOPPED`.

Alors qu'en Java les modifications ont lieu dans un *system thread* qui a pour effet de suspendre *tous* les threads en cours d'exécution avant de commencer à s'exécuter lui-même, on dispose ici d'un mécanisme permettant de suspendre localement la portion de l'application qui doit être modifiée. Arrêter un composant ne nous conduit pas à suspendre la totalité de l'application.

### Instanciation

Les opérations fournies par les interfaces des contrôleurs nous permettent d'utiliser, d'observer, de configurer et de reconfigurer les composants existants. Dans le modèle Fractal, on définit également des opérations pour créer de nouveaux composants. Ceux-ci sont créés par d'autres composants qu'on appelle les factories. Les factories qui peuvent créer plusieurs types de composants sont dites génériques. Une factory, générique ou standard, fournit l'opération `newFcInstance()` pour créer une nouvelle instance (ou éventuellement renvoyer sa référence comme dans le cas d'un *design pattern* de type singleton).

Si les composants sont créés par les factories qui sont elles-mêmes un type particulier de composant, il est légitime de se demander comment on crée une factory. Si les factories sont elles-mêmes créées par d'autres factories, alors nous sommes confrontés à un phénomène de récursivité sans fin. Pour

s'en sortir, il existe un composant particulier : le bootstrap. Celui-ci n'a guère besoin d'être créé. Il existe dès l'initialisation du système et est accessible par convention. On peut obtenir sa référence grâce à l'opération `getBootstrapComponent()`.

#### Système de typage

Dans le système de typage de Fractal, un type de composant est un ensemble de types d'interfaces. L'opération `getFcInterfaceTypes()` invoquée sur un type de composant permet de récupérer la liste de ses types d'interfaces. Sur chaque type d'interface, on peut récupérer le nom, la signature, un booléen qui indique s'il s'agit d'une interface client ou serveur, etc. Les types de composants et les types d'interfaces doivent être créés par des types de factories, qui fournissent les deux opérations à cet effet. L'opération permettant de créer des types de factories se trouve dans l'interface du composant bootstrap.

Les interfaces `ComponentType`, `InterfaceType` et `TypeFactory` étendent toutes trois l'interface `Type`. La seule opération de cette dernière est `isFcSubTypeOf(Type type)`. Cette relation fournit une relation suffisante (mais non nécessaire) de substituabilité. Si $T_1$ est un sous-type de $T_2$, alors $T_1$ peut remplacer un composant de type $T_2$ dans n'importe quel contexte, puisque $T_1$ a possède les mêmes interfaces que $T_2$ en plus de celles qui lui sont spécifiques. Le même genre de raisonnement peut s'appliquer aux types d'interfaces.

Remarquons qu'un type n'est pas modifiable *en soi*. Cependant, on peut en créer. Si on veut ajouter un type interface à un type de composant, il faut créer un nouveau type de composant à partir de celui que l'on souhaiterait modifier. Il n'est pas non plus possible de changer le type d'une instance de composant à chaud.

### 3.4.2 Exemple de reconfiguration

Nous partons de la situation décrite dans la figure 3.8. L'application est représentée par le composant `root` qui fournit une interface `m`. Il comprend deux sous-composants : un client et un serveur, qui communiquent via les interfaces client et serveur toutes deux dénommées `s`. Le composant `root` est le seul à implémenter le `ContentController` (`CC`) puisqu'il est le seul à être composé. Ses deux sous-composants sont dit primitifs.

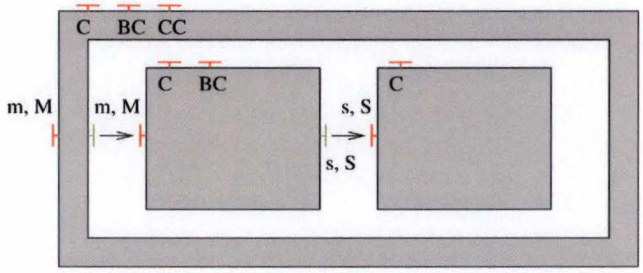


FIG. 3.8 – Exemple de système modélisé en Fractal. Le système `root` (abrégé `r`) comprend deux sous-composants : un client `c` qui fournit les interfaces `Component` et `BindingController` et un serveur `s` qui fournit l'interface `Component`. Source : [4]

La reconfiguration va consister à remplacer le composant serveur. Nous supposons ici que les opérations du contrôleur du cycle de vie sont récursives (car, rappelons-le, ceci n'est pas imposé par le modèle). Pour commencer, on arrête le composant `root`. Récursivement, ses deux sous-composants sont également arrêtés.

```
r.getFcInterface("lifecycle-controller").stopFc();
```

On récupère ensuite la référence du composant serveur `s`. D'abord, puisqu'on ne dispose au départ que que de la référence de `root`, on passe par `m` pour récupère la référence de `c`. Ensuite, à partir du `BindingController` de `c`, on récupère la référence du serveur en passant par les interfaces `s`.

```
Component c = r.getFcInterface("binding-controller").
      lookupFc("m")).getFcItfOwner();

Component s = c.getFcInterface("binding-controller").
      lookupFc("s")).getFcItfOwner();
```

A ce stade, on peut délier les deux sous-composants et supprimer `s`.

```
c.getFcInterface("binding-controller").unbindFc("s");
r.getFcInterface("content-controller").removeFcSubComponent(s);
```

On crée ensuite une nouvelle instance de sous-composant serveur, à partir de la factory désignée par `gf`. L'opération utilise trois paramètres :

- `sType` désigne le type du composant.
- `primitive` indique que le composant n'aura pas de sous-composant. Autrement dit, cela indique qu'il n'implémente pas le contrôleur de contenu.
- `NewSImpl` référence le contenu du composant. Ici, cela renvoie à son implémentation.

```
Component newS = gf.newFcInstance(sType, "primitive", "NewSImpl");
```

Il reste à ajouter le nouveau sous-composant au composant paretn `root`, à le relier au composant `c` et à redémarrer le `root`.

```
r.getFcInterface("content-controller").addFcSubComponent(newS);
c.getFcInterface("binding-controller").bindFc("s", newS);
r.getFcInterface("lifecycle-controller").startFc();
```

### 3.4.3 Discussion

Fractal permet de choisir, pour chaque composant, s'il implémente les contrôleurs. Mais on ne peut ici séparer l'observabilité et la reconfiguration. Dans le `ContentController` par exemple, on trouve l'opération `getFcSubComponents()` qui permet d'obtenir la liste des composant et `addFcSubComponent()` qui permet d'en ajouter.

De plus, les interfaces fonctionnelles (celles écrites par les développeurs des applications, propres au domaine d'application) sont placées sur le même plan que les interfaces des contrôleurs. On imagine la confusion que cela peut induire lors du développement. Dans les deux solutions précédentes à l'inverse, le code de base de l'application est séparé des opérations de reconfiguration. Dans une application C par exemple, qui contient tout le code métier, l'introduction des aspects - le tissage - s'effectue via une commande qui prend en paramètre l'identifiant du processus de l'application et le répertoire où sont se trouve le code des aspects.

### Caractéristiques d'analyse

**Granularité :** Le concept central du modèle est le composant. On peut en créer des instances via une factory en précisant en paramètre le type à instancier. On peut aussi les supprimer, alors que les deux solutions précédentes sont purement additives. Si l'on peut modifier leur contenu (ajout et suppression de sous-composants, modifications des liens entre les sous-composants), ni le type ni l'implémentation ne peuvent être transformés directement. Il demeure néanmoins possible de créer dynamiquement de nouveaux types, mais sans moyen explicite pour pouvoir en modifier ou en supprimer.

Le modèle peut, certes, être étendu mais en omettant d'inclure une opération pour modifier un type, la question de savoir si les instances de composants devraient être affectées aussitôt reste en suspend. De même, si on modifie un type en supprimant une interface, que faire si des connexions avec des instances de cette interface sont déjà établies ?

**Type de modification :** Dans les modèles orientés-composant, par définition, c'est la structure qui prime. Les principales opérations de transformation se basent donc sur les principes structuraux : ajout ou suppression de composant, modification d'une connexion, etc. Cependant, des modifications comportementales ont bien lieu (voir la reconfiguration donnée en exemple plus haut, qui consiste précisément à remplacer une instance de composant par une instance identique si ce n'est qu'elle possède une implémentation différente), mais elles sont implicites. Dans les deux solutions précédentes, l'ordre des deux types de modification étaient inversés : en C comme en Java, c'étaient les modications au niveau du code source qui étaient explicites qui pouvaient éventuellement résulter dans des changements de structure.

**Ouverture :** Lors du design d'un système basé sur le modèle de composants Fractal, on doit décider quels composants possèdent quelles interfaces. On doit donc décider *a priori* si un composant donné est reconfigurable ou pas, et dans quelle mesure il l'est. Selon les interfaces implémentées, on peut entrevoir une marge de manoeuvre ultérieure plus ou moins large. Nous voyons que selon les objectifs du modélisateur, les capacités de reconfiguration doivent être pensées *a priori*. On peut donc conclure que Fractal est une solution partiellement ouverte.

**Gestion de l'état :** Les opérations du contrôleur de cycle de vie permettent d'arrêter « proprement » la portion de l'application concernée par une modification dynamique. La norme cependant, volontairement abstraite et minimaliste, ne précise pas le comportement d'un appel qui a lieu sur un composant arrêté, sur une interface client qui n'est pas branchée ou encore dans le cas où l'appel est encore en cours au moment où l'on arrête ou supprime une instance de composant ou lorsque l'on change une connexion entre deux instances de composant. Selon les choix d'implémentation, l'appel peut renvoyer une exception, des messages peuvent être perdus ou l'application peut crasher. La norme de nous dit rien non plus quant au sujet des transferts d'état entre instances.

## 3.5 OpenCOM

OpenCOM est un modèle de composants léger, réflexif, indépendant du langage qui l'implémente, qui se base sur les concepts essentiels de Microsoft COM. Les caractéristiques de plus haut niveau de COM telles que la distribution, la persistance, la sécurité et la gestion de transactions ne sont pas reprises dans OpenCOM. Ce modèle, ainsi que le concept de framework de composant, sont les deux technologies à la base de l'implémentation de l'intergiciel OpenORB v2.

### 3.5.1 Le modèle de composants OpenCOM

L'architecture est bâtie autour du composant `OpenCOM` (figure 3.9) qui offre un environnement d'exécution standard disponible dans chaque espace d'adressage OpenCOM. Le rôle du singleton `OpenCOM` est de gérer un répertoire de types de composants disponibles ainsi que de procéder aux créations et suppressions d'instances. Son interface `IOpenCOM` sert de point central aux requêtes de reconfiguration (créations et suppressions d'instances et connexions et déconnexions entre composants) dans l'espace d'adressage.

Afin de faciliter la reconfiguration, le composant `OpenCOM` enregistre chaque opération de création d'instance, de connexion, etc. dans le *system graph* (figure 3.9). Il s'agit d'une méta-structure qui représente l'état de l'application. Le *system graph* permet également d'obtenir de l'information introspective, toujours via l'interface `IOpenCOM` qui fournit pour cela les opérations `getConnectionInfo()`, `enumComponents()`, etc.

Les autres composants peuvent avoir des interfaces particulières ( *Custom Interface* sur la figure 3.9). Cependant, pour être conforme au modèle OpenCOM, chaque composant doit implémenter deux interfaces de gestion de composants destinées à être utilisées par le singleton `OpenCOM` : les interfaces `IReceptacles` et `ILifeCycle`. La première fournit les opérations permettant de gérer les connexions avec d'autres composants (le mot *receptacle* est utilisé ici pour qualifier les interfaces client) tandis que la seconde, similairement à celle de Fractal, fournit les opérations `startup()` et `shutdown()`. Ces deux interfaces ne sont utilisées que par le composant `OpenCOM`.

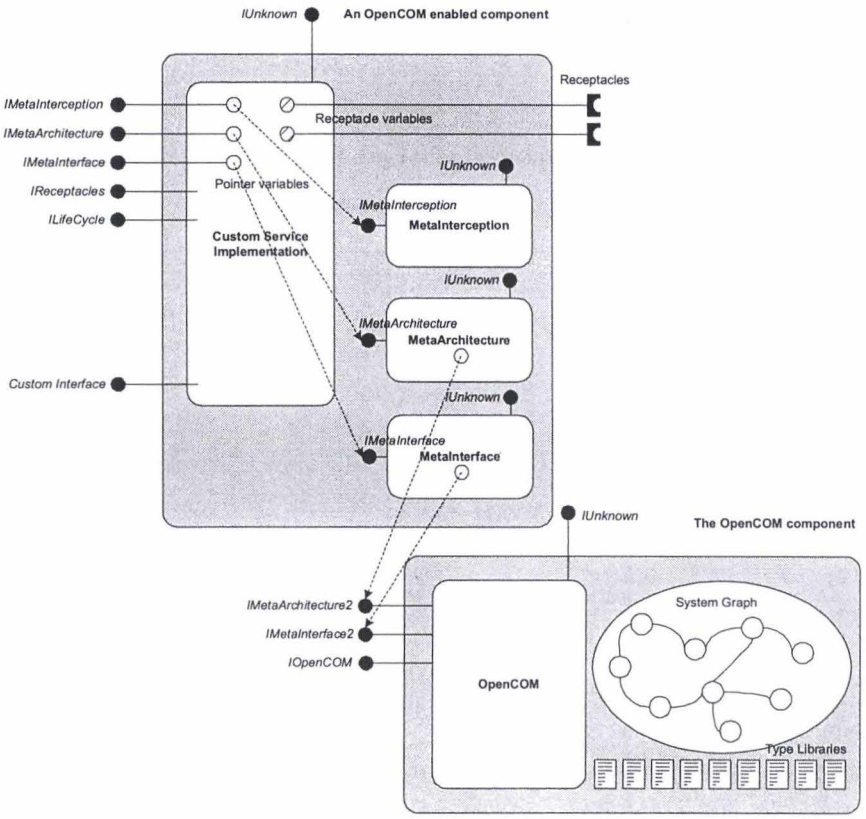


FIG. 3.9 – Chaque système OpenCOM comprend au moins le composant `OpenCOM` (en bas) qui gère les créations et suppressions d'instances ainsi que les connexions entre interfaces. Les interfaces clients sont appelées réceptacles tandis que les interfaces serveur sont simplement appelées interfaces ou *sink's interfaces*. Chaque composant implémente les interfaces `IReceptacles` et `ILifeCycle` utilisées par le composant `OpenCOM` et comprend trois sous-composants qui sont `MetaInterception`, `MetaArchitecture` et `MetaInterface`. Source : [11, 10]

De plus, chaque composant OpenCOM possède trois sous-composants. Ces interfaces implémentent des opérations réflexives détaillées plus loin[6] :

- `MetaInterception`
- `MetaArchitecture`
- `MetaInterface`

[6]L'API complète se trouve en annexe.

### IUnknown

Cette interface joue trois rôles. Tout d'abord, elle sert d'identifiant et de référence pour chaque composant. Si l'on demande par exemple au singleton `OpenCOM` la liste des composants du système, il renvoie la liste des références vers les interfaces `IUnknown` de chaque composant. Ensuite, elle fournit un mécanisme de découverte dynamique des interfaces. L'opération `QueryInterface()` peut être invoquée sur chaque composant pour savoir s'il implémente telle ou telle interface, auquel cas l'opération renvoie la référence de l'interface. Cela permet de déterminer si deux pointeurs référencent le même composant, en comparant les résultats de l'appel de l'opération `QueryInterface()`. Notons que les autres interfaces étendent celle-ci.

### IReceptables

Cette interface n'est appelée que par le singleton `OpenCOM` pour gérer les connexions entre composants. Elle comprend les deux opérations : `connect()` et `disconnect()`.

Cette interface est comparable au `BindingController` de Fractal mais ne peut fournir d'information introspective (comme la liste des connexions en cours). Les capacités introspectives sont accessibles via l'interface `IMetaArchitecture`, détaillée plus bas.

Il existe plusieurs « styles » de réceptacles :

- Receptacle ***single pointer :*** ce réceptacle contient un seul pointeur vers une interface. C'est le style de receptacle le plus commun.

- Receptacle ***multi pointer :*** ce réceptacle contient plusieurs pointeurs vers des implémentations du même type d'interface, sans préférence.

- Receptacle ***muti pointer with context :*** ce réceptacle contient plusieurs pointeurs vers des implémentations du même type d'interface. L'interface est choisie en passant de l'information contextuelle lors de l'invocation d'une méthode sur le receptacle. Ce style est utilisé dans les implémentations utilisant des CFs (*component framework*) où il est nécessaire de choisir parmi plusieurs plug-ins (détaillé plus bas).

Les réceptacles disposent d'un mécanisme optionnel de verrou (*lock*). Si l'on utilise ce mécanisme, les couches logicielles de plus haut niveau construites

sur le modèle peuvent se fier à OpenCOM pour maintenir l'intégrité au niveau des invocations ; intégrité mise à mal par la concurrence entre les threads qui émettent des invocations sur le réceptacle et le composant `OpenCOM` qui effectue des reconfigurations qui impliquent le réceptacle. Lorsque l'on supprime un composant, tous les réceptacles qui sont en connexion avec les interfaces de ce composant doivent être verrouillés et le demeurer jusqu'à être explicitement déverrouillés, par exemple lorsqu'ils sont reconnectées à d'autres interfaces, lorsque la reconfiguration est terminée.

Le mécanisme de verrou nécessite un comptage de références pour connaître le nombre d'invocations en cours sur un réceptacle. Le composant `OpenCOM` ne peut obtenir le verrou que si ce compteur est égal à zéro. Chaque invocation sur un réceptacle est précédée d'une vérification du statut du verrou. Si le réceptacle est verrouillé, l'invocation renvoie un code d'erreur. Si ça n'est pas le cas, le compteur d'invocations est incrémenté de un et décrémenté quand l'appel se termine.

Cette politique de gestion des appels en cours n'est pas sans rappeller la stratégie 2 décrite plus haut pour une transformation de méthode en Java (p. 28). Dans cette dernière, on doit attendre un moment où il n'y a plus d'appel en cours. A cet instant seulement, on peut placer tous les threads du système dans l'état *suspend* afin de procéder au remplacement de la méthode. Le principal avantage de cette stratégie est qu'elle garantit que deux versions de la même méthode ne peuvent coexister. En contrepartie, attendre qu'il n'y ait plus aucune invocation en cours peut être long, et la probabilité d'atteindre un tel état est encore plus faible dans les applications multi-threading. Le mécanisme de verrou est néanmoins plus efficace, puisqu'il empêche de nouvelles invocations. Le risque qu'un appel ne se termine jamais existe toujours, mais il est moins probable dans une architecture orientée-composant que dans une architecture orientée-objet.

### ILifeCycle

Comme la précédente, cette interface ne peut être appellée que par le composant `OpenCOM`. Son rôle est similaire à l'interface `LifeCycleController` de Fractal. Elle fournit les opérations `startup()` et `shutdown()`. Ces opérations jouent le rôle de constructeur et de destructeur qu'on retrouve habituellement dans l'orienté-objet. A l'inverse de Fractal, la sémantique d'OpenCOM est un peu plus précise, puisqu'elle impose que l'état interne puisse être récupéré quand un composant est redémarré. Autrement dit, l'opéra-

tion `shutdown()` équivaut à une suspension (éventuellement temporaire, si celui-ci n'est pas supprimé) de l'activité du composant.

L'opération `startup()` permet d'initialiser le composant. On distingue ici le temps de la création et celui de l'initialisation, parce que l'opération `startup()` invoque des opérations sur des réceptacles qui doivent être connectés après que l'instance ait été créée par le composant `OpenCOM` avant l'initialisation, c'est-à-dire avant l'initialisation. L'opération `startup()` devrait être appelée dès que l'instance a été créée. De la même façon, l'opération `shutdown()` permet à une instance de réagir à sa suppression.

En d'autres mots, le cycle de vie d'une instance se déroule selon ces différentes étapes :

1. Création par le composant `OpenCOM`
2. Initialisation de l'instance (connexions des réceptacles, etc.)
3. Démarrage de l'instance via l'opération `startup()`
4. Vie de l'instance pendant laquelle peuvent survenir plusieurs phases d'arrêt et de redémarrage
5. Arrêt définitif de l'instance
6. Suppression de l'instance par le composant `OpenCOM`

### IMetaInterception

Cette interface permet au programmeur d'associer (et de dissocier) des intercepteurs avec des interfaces. Les intercepteurs sont des composants auxquels on associe des pré- et post-méthodes. Elles sont exécutées avant ou après (ou les deux) chaque invocation sur les interfaces auxquelles elles sont associées. Plusieurs intercepteurs peuvent être associés à une interface. Ils peuvent être supprimés ou réorganisés à chaud.

L'interface `IMetaInterception` comprend les opérations `createInterceptor()` et `deleteInterceptor()`. L'interface `IInterceptor` comprend les opérations `addPreMethod()`, `delPreMethod()`, etc. ainsi que deux opérations introspectives permettant de consulter les listes des pré- et des post-méthodes associées à l'intercepteur.

OpenCOM introduit un mécanisme de reconfiguration comparable aux technologies orientées-aspect évoquées plus haut ou au mécanisme de trigger utilisé en base de données. Ce mécanisme est cependant moins riche que la technologie orientée-aspect puisqu'il ne permet pas de remplacer le code existant pour une méthode donnée, mais seulement de le faire précéder et/ou succéder d'une portion de code supplémentaire. De plus, on ne peut sélectionner certaines méthodes particulières parmi celles de l'interface : un intercepteur est exécuté pour chaque appel de chaque méthode sur l'interface concernée.

#### IMetaArchitecture

Cette interface permet d'obtenir des informations sur les connexions établies entre les composants. L'opération `enumConnsToInft()` renvoie une liste d'identifiants de connexion (qui peuvent par exemple être utilisés en paramètre d'entrée pour les opérations de suppression adressées à l'interface `IOpenCOM`) de toutes les connexions établies sur une interface donnée. L'opération `enumConnsFromRecp()` renvoie la liste des identifiants de connexion à partir d'un réceptacle donné. Cette interface est similaire aux capacités introspectives du `BindingController` de Fractal.

#### IMetaInterface

Cette interface fournit des opérations introspectives permettant d'obtenir de l'information sur les types d'interface déclaré par le composant visé.

### 3.5.2 Component Frameworks

L'intergiciel OpenORB v2 repose sur deux technologies qui sont d'une part le modèle de composants OpenCOM, et d'autre part le concept de framework de composants, initialement défini par Szyperski [40] comme « une collection de règles et d'interfaces qui régissent les interactions d'un ensemble de composants qui y sont liés. » Ils maintiennent des contraintes et des règles spécifiques à un domaine d'application ou un environnement d'exécution particulier et permettent de maintenir l'intégrité lors des opérations de reconfiguration.

Les component frameworks (ou CFs) peuvent se voir intégrer des plug-ins (c'est-à-dire des sous-CFs) implémentés par des composants OpenCOM, qui modifient leur comportement et qui sont acceptés ou refusés sur base des contraintes qui y sont définies. Il est possible de hiérarchiser des CFs pour

mieux contrôler la portée et les effets d'une reconfiguration. Afin de maintenir l'intégrité et de réduire le couplage, les CFs interdisent à un composant de dépendre d'un autre composant qui serait en dehors du CF. Les CFs fournissent un environnement de propriétés architecturales bien définies et les invariants pour leurs plug-ins. Ils simplifient le développement des composants par la réutilisabilité. Ils augmentent la compréhension et l'évolutivité d'un système.

Un pattern *manager/managed* est introduit afin de maintenir l'intégrité et de mieux séparer les préoccupations entre les fonctionnalités « courantes » du intergiciel et les opérations de reconfiguration. Un CF est un *manager* alors que ses plug-ins sont les *managed*. De la même façon, les sous-CFs sont vus comme des *managed* pour les CFs parents.

La figure 3.10 nous montre que la structure d'OpenORB est elle-même un CF qui contraint les interactions entre ses sous-CFs qui le constitue et gère leur reconfiguration. Le CF `media streaming` autorise des filtres implémentés comme des plug-ins. Il gère le graphe qui représente la configuration des filtres et offre une méta-interface pour la reconfiguration de ce graphe. En interne, les composants qui implémentent les filtres fournissent une interface *managed* au `media streaming`. Ces composants acceptent des requêtes du CF parent et peuvent émettre des événements. Par exemple, un filtre qui n'a pas les ressources nécessaires pour s'exécuter avec le flot de données qu'on lui transmet peut émettre un événement pour prévenir son CF parent. Si ce dernier est en mesure de traiter l'événement, il peut par exemple réagir en diminuant le flux d'entrée dans le filtre. Mais s'il ne peut comprendre l'événement, il le transmet lui-même à son CF parent, autorisant les composants externes à traiter l'événement correctement (par exemple, en ajoutant un nouveau filtre dans le graphe). Le CF utilise également sa connaissance implicite du domaine d'application (*domain-specific knowledge*) pour mieux gérer les opérations de reconfiguration. Dans le cas présent, il pourrait bufferiser de le flux de données pendant les opérations de reconfiguration.

OpenORB est structuré comme un CF lui-même composés de trois couches abritant des CFs (figure 3.10). Chaque sous-CF n'est autorisé à accéder qu'aux interfaces de sa couche ou des couches inférieures. Le CF principal gère le cycle de vie et les liaisons entre les ses sous-CFs. De plus, il impose cette règle de composition en trois couches.

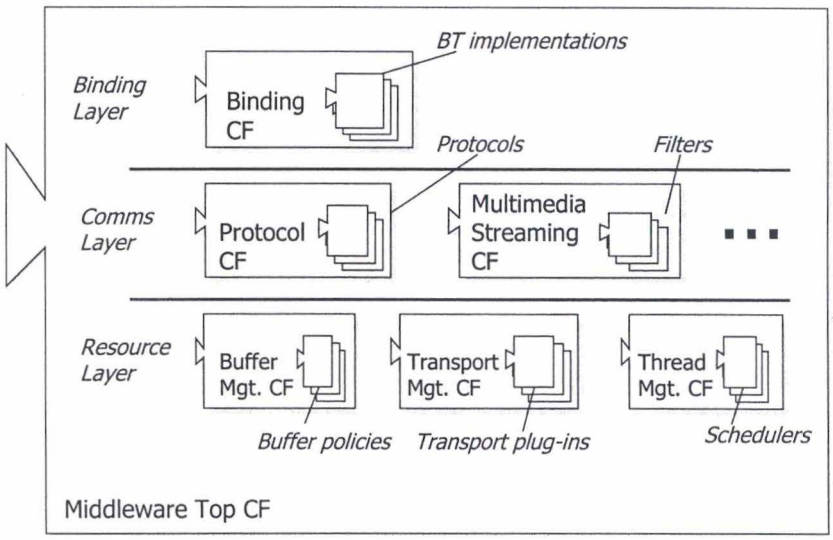


FIG. 3.10 – Les CFs sont représentés par des blocs auxquels on adjoint un petit trapèze. L'architecture OpenORB est construite comme un CF. Celui maintient une règle de division des sous-CFs en trois couches. Source : [11]

La figure 3.10 nous montre une configuration initiale de l'intergiciel. Celui-ci peut être modifié dynamiquement par l'ajout de nouveaux CFs, dans la mesure où ces nouveaux composants sont compatibles avec les contraintes définies au niveau du CF principal. De même au niveau des sous-CFs, telle ou telle couche peut être améliorée par l'ajout de nouveaux composants. De la même façon, les CFs de plus bas niveau peuvent se voir intégrer des plug-ins.

### 3.5.3 Discussion

Nous remarquons tout d'abord une distinction plus claire entre introspection et reconfiguration. Le rôle du `BindingController` de Fractal est ici partagé entre l'interface `IMetaArchitecture` pour obtenir de l'information sur les connexions établies et l'interface `IReceptacles` pour transformer ces connexions.

Ensuite, le concept de framework de composants, couplé avec le modèle OpenCOM, introduit l'idée d'une gestion de l'évolution en adéquation avec un domaine particulier (*domain-specific knowledge*).

De plus, la granularité s'étend sur plusieurs directions. Il est possible de charger de nouvelles instances de composants et de modifier les connexions

qui leur permettent de communiquer, et il est également possible d'associer et de dissocier des pré- et post-méthodes à une interface.

Cependant, bien que le modèle OpenCOM se décrive comme étant « hiérarchiquement composable », les mécanismes permettant de gérer les sous-structures sont assez flous. A l'inverse de Fractal, nous ne trouvons guère ici au sein des interfaces d'opérations susceptibles de modifier ou même de lire la structure interne d'un composant. Du reste, en OpenCOM, une configuration donnée correspond à un ensemble d'instances et un ensemble de connexions. Les types de composants sont stockés dans le singleton `OpenCOM` (figure 3.9) et il ne semble pas possible d'en charger de nouveaux.

Enfin, si dans le modèle Fractal, on insiste particulièrement sur le fait que toutes les interfaces sont à option, l'alternative n'est pas aussi claire dans le modèle OpenCOM. Tout système qui l'emploie comme base doit-il implémenter toutes ses interfaces ?

### Caractéristiques d'analyse

**Granularité :** Le modèle autorise la création et la suppression d'instances de composant, la modification de connexions entre composants, ainsi que l'ajout et la suppression de pré- et post-méthodes associées à une interface. Le modèle n'offre guère d'opération pour l'ajout, la suppression ou la modification de types.

**Type de modification :** A l'instar de Fractal, le modèle est basé sur le concept de composant et c'est sur ce concept même que l'on procède aux opérations de reconfiguration. Les opérations dans leur ensemble sont donc - tout comme en Fractal - de nature structurelle (modification d'une connexion, ajout d'instances, etc.). Cependant, OpenCOM permet également l'ajout et la suppression de code au niveau des interfaces, mécanisme proche de l'orienté-aspect. Ces modifications sont plutôt de nature comportementale. On peut donc conclure qu'OpenCOM autorise les deux types de modifications, via des interfaces distinctes.

**Ouverture :** Puisqu'il n'est guère possible apparemment de charger à chaud de nouveaux types, il faut prévoir ceux dont on aura besoin. Nous sommes donc ici dans le cas d'un système partiellement ouvert. Le système de plug-ins des CFs confirme cette classification. En effet, les CFs permettent de préciser un cadre pour décider si un plug-in donné est compatible avec la politique du CF parent.

**Gestion de l'état :** OpenCOM est un peu plus précis que Fractal sur ce point. Outre les opérations permettant d'arrêter et de redémarrer une instance (dont la sémantique est également plus précise en OpenCOM), il prévoit un mécanisme de verrou au niveau des réceptacles. Le modèle OpenCOM n'aborde pas directement le problème d'un transfert d'état entre instances, mais précise que de tels mécanismes, s'ils sont implémentés, devraient se trouver au niveau des CFs.

## 3.6 Tableau récapitulatif

| | **Arachne** | **PJama** | **Fractal** | **OpenCOM** |
|---|---|---|---|---|
| **Granularité** | Les fonctions et les accès aux variables globales peuvent être utilisés comme joinpoints. Les modifications sont seulement additives. | Méthodes et classes, dans les limites du *binary compabible*, c'est-à-dire des modifications globalement additives. | Modification du contenu, création et suppression d'instances de composant, création de types de composant. | Création et suppression d'instances de composant, modification de connexions entre composants. Ajout et suppression de pré- et post-méthodes associées à une interface. Pas d'opération pour l'ajout, la suppression ou la modification de type. |
| **Type de modification** | Comportementales | Comportementales (méthodes) et structurelles (classes) | Structurelles | Structurelles et comportementales (pour les pré- et post-méthodes) |
| **Ouverture** | Ouvert (partiellement ouvert pour $\mu$Dyner) | Ouvert | Partiellement ouvert, puisque chaque composant peut implémenter les interfaces que l'on souhaite, selon les besoins d'évolution ultérieure. | Partiellement ouvert, puisqu'il semble impossible de charger de nouveaux types. Les CFs exigent de définir une politique qui accepte ou refuse les plug-ins. |
| **Gestion de l'état courant** | Les appels et accès aux variables qui précèdent la modification se terminent (éventuellement) sur l'ancienne version | Stratégie 4 : Les nouveaux appels ont lieu sur la nouvelle version. L'ancienne version est supprimée quand les appels en cours se terminent. Au niveau structurel, mécanisme de conversion personnalisée des instances | Opérations du cycle de vie pour maintenir l'intégrité | Opérations du cycle de vie, verrous sur les réceptacles, possibilité d'implémenter le transfert d'état au niveau des CFs |

# Chapitre 4

# Une autre approche de la reconfiguration dynamique

Bracha et Ungar [3] défendent l'idée selon laquelle les opérations introspectives, qui appartiennent à un méta-niveau, devraient être implémentées séparément. Or, dans la plupart des langages orientés-objet, les opérations du niveau de base et du méta-niveau sont côte à côte. Comme on le voit dans l'exemple de la figure 3.5 (p. 42), les capacités d'introspection sont au même niveau d'abstraction que le code de l'application. Dit autrement, si l'on demande l'interface d'une classe donnée, on trouve tout autant des méthodes de type métier que des méthodes fournissant des méta-informations (`getField()`, `getMethod()`, `getModifiers()`, `getPackage()`, `getName()`, etc.).

Ils défendent la séparation entre le code métier du niveau de base et les opérations introspectives qui appartiennent à un méta-niveau en vertu de deux principes : l'encapsulation et la stratification. L'encapsulation, tout d'abord, nous dit qu'un module ne peut dépendre des particularités de l'implémentation d'autres modules. En d'autres mots, les fonctionnalités de méta-niveau doivent encapsuler leur propre implémentation. Ce principe tend à séparer l'interface donnant accès à des fonctionnalités de méta-niveau d'une implémentation particulière. En second, la stratification se base sur l'idée de placer les méta-fonctionnalités dans un sous-système séparable. Ce principe exprime le souhait de ne pas devoir utiliser l'introspection ou la reconfiguration si cela n'est pas nécessaire. En effet, dans le domaine des logiciels de vol pour satellites comme dans d'autres, l'on peut souhaiter que le système se trouve dans un état tel qu'il ne peut être ni introspecté ni reconfiguré lorsqu'il est en exploitation, mais qu'il puisse éventuellement le devenir lors des phases de maintenance.

Des problèmes similaires se posent en Fractal. Le modèle utilise le concept d'interface fonctionnelle pour introduire des points de communication entre composants. Ces interfaces sont écrites par les modélisateurs et comprennent la logique métier. Or, les interfaces des contrôleurs sont mises sur le même plan que ces interfaces fonctionnelles écrites par les modélisateurs.
Le `ContentController`, par exemple, permet d'obtenir la liste des sous-composants.

En outre, si en Java les opérations d'introspection sont au même niveau que la programmation métier (package `java.lang.reflect`), les opérations de reconfiguration (ajouter de champs, méthodes, etc.) sont séparées, puisqu'elles se déroulent dans le cadre du debugger. En Fractal par contre, les opérations d'introspection et les opérations de reconfiguration se trouvent dans les mêmes interfaces. A l'exemple de l'interface du `ContentController` :

```
interface ContentController {
   any[] getFcInternalInterfaces();
   any getFcInternalInterface(string itfName);
   Component[] getFcSubComponents();
   void addFcSubComponent(Component c);
   void removeFcSubComponent(Component c);
}
```

On trouve à la fois des opérations d'introspection (`getFcInternalInterfaces()`, `getFcSubComponents()`) et des opérations de reconfiguration (`addFcSubComponent(Component c)`). Or, dans certains cas et pour diverses raisons, on voudrait parfois vouloir construire un système seulement introspectif ou seulement reconfigurable.

En outre, le succès récent des DSL (*domain-specific languages*), qui impliquent le développement de langages de modélisation propres à chaque domaine d'application (voir la section 4.1.1), nécessiterait de devoir repenser le modèle d'introspection pour chaque DSL. L'approche développée dans ce chapitre tend vers plus de généricité.

Finalement, nous constatons que dans toutes les solutions étudiées jusqu'à présent, la granularité est limitée. En Fractal, si l'on peut ajouter et supprimer des composants, on ne peut pas supprimer ou modifier un type ou une interface. Dans OpenCOM, on ne peut apparemment que créer et supprimer des instances de composant et des pré- et post-méthodes. Notre approche propose de généraliser la granularité à tous les éléments d'un modèle.

L'approche décrite dans ce chapitre synthétise le travail réalisé par les chercheurs de Telecom Bretagne, dans le cadre du projet SP@CIFY, afin de développer des mécanismes de reconfiguration dynamique adaptés, pendant la durée du stage, c'est-à-dire entre septembre 2007 et janvier 2008.

Nous commencerons par introduire deux notions essentielles sur lesquelles repose l'approche : le *domain-specific modeling* et la notion de métamodèle.

## 4.1 Notions introductives

### 4.1.1 Domain-specific modeling

Le *domain-specific modeling* (ou DSM) est une méthodologie d'ingénierie logicielle impliquant l'usage systématique d'un *domain-specific language* (ou DSL) souvent graphique afin de représenter différents aspects d'un système. Les DSL se distinguent des langages de modélisation et de programmation classiques, dit *general purpose* (Java, C, etc.) conçus pour n'importe quel domaine d'application, par un niveau d'abstraction plus élevé qui entraîne une facilité d'utilisation et un moindre niveau de détails, en prenant en compte les concepts d'un domaine d'application particulier, en préservant les ontologies utilisées dans les domaines experts, permettant de capitaliser les connaissances acquises dans un domaine particulier.

Le langage YACC par exemple, est un DSL conçu pour créer un parser à partir de la description d'une grammaire abstraite, en permettant de s'abstraire des détails techniques. Il existe encore d'autres exemples comme CSound, qui permet de créer des applications de gestion sonore à partir de scripts.

Le DSM contient souvent l'idée de la génération de code, c'est-à-dire l'idée d'automatiser la création de code exécutable directement à partir de modèles, permettant ainsi d'augmenter de façon impressionnante la productivité du développeur et d'améliorer la qualité du code généré.

### 4.1.2 Métamodélisation

Un métamodèle se caractérise par un ensemble de règles, de concepts, de contraintes afin d'écrire des modèles dans un domaine d'application donné. Le modèle est une abstraction d'un phénomène dans le monde réel. Le métamodèle se situe plus haut dans l'abstraction, puisqu'il décrit les propriétés

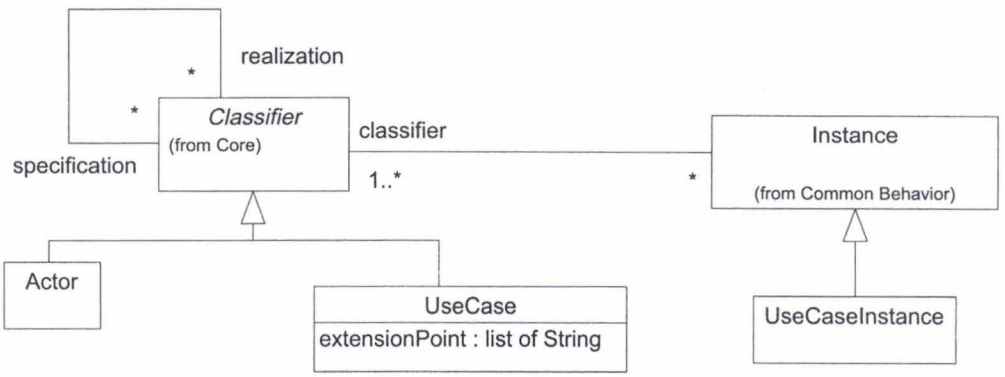


FIG. 4.1 – Métamodèle des diagrammes de cas d'utilisation UML : il s'agit d'un type de modèle particulier qui représente les règles, les concepts et les contraintes nécessaires à l'écriture des modèles de cas d'utilisation. Tous les *Use Cases Diagrams* possibles peuvent être vu comme des instances de ce métamodèle.

des modèles eux-mêmes. On peut dire qu'un modèle est conforme à un métamodèle de la même façon d'un programme écrit dans un langage donné est conforme à la grammaire de ce langage.

A titre d'exemple, le formalisme des diagrammes des cas d'utilisation d'UML (ou *Use Cases Diagrams*), principalement utilisé pendant les phases d'analyse des besoins, permet de représenter sous forme de modèles les fonctionnalités attendues d'un système et le comportement de ce système en faisant abstraction de sa structure interne.

Les deux concepts principaux sont les **acteurs** et les **cas d'utilisation**. L'acteur représente un utilisateur du système tandis que le cas d'utilisation représente une fonctionnalité que l'acteur peut utiliser. Chaque cas d'utilisation spécifie une séquence d'actions et représente une partie du comportement du système. Les instances de cas d'utilisation et d'acteurs interagissent quand un service du système est utilisé. Ce formalisme permet donc d'écrire des modèles simples, représentant les fonctionnalités essentielles d'un système, de cerner les groupes d'utilisateurs pertinents et de définir les interactions entre les utilisateurs et les fonctionnalités, c'est-à-dire de déterminer pour un utilisateur donné à quelles fonctionnalités il a accès.

Les règles, les concepts et les contraintes de ces modèles peuvent être modélisé dans un type particulier de modèle : un métamodèle, comme celui représenté à la figure 4.1. Le métamodèle définit formellement les liens entre les différents concepts que l'on peut utiliser pour créer un ensemble de modèles. On voit par exemple la relation de spécialisation qui existe entre les acteurs. Un acteur donné peut hériter de plusieurs autres acteurs (et a donc accès aux mêmes fonctionnalités que ces acteurs parents) et peut également posséder plusieurs sous-types d'acteurs.

Notons cependant que la terminologie est parfois un peu ambiguë. On a souvent tendance à appeler modèle ce qui pourrait en fait être exprimé par un métamodèle, dont les modèles seraient les instances.

## 4.2 Une autre approche

Le but de l'approche détaillée ici est de créer une base sur laquelle on puisse, pour chaque domaine d'application, engendrer un *domain-specific metamodel* dont on pourrait plus tard déduire automatiquement un ensemble d'opérations de reconfiguration. Le domaine visé est initialement celui des logiciels de vol pour satellites, mais l'approche peut être généralisée à tout domaine nécessitant l'usage d'un DSMM.

Pour ce faire, on a d'abord un métamodèle très général reprenant les principaux aspects de la modélisation : les concepts du modèle, leurs relations, les contraintes, les règles, etc. Pour plus de facilité, ce métamodèle sera présenté en plusieurs parties (dans la section 4.2.1), chaque morceau représentant un type de propriété de métamodèle dont certaines sont que l'on retrouve dans de nombreux DSMM (décomposition hiérarchique, capacités d'assemblages, capacités de réfléxivité, etc.). Ils correspondent au niveau supérieur de la figure 4.2.

A partir de ces métamodèles, on sélectionne les aspects dont on a besoin pour créer le DSMM, et on les fusionne. Ce DSMM, représenté au niveau intermédiaire sur la figure 4.2, permettra de créer des modèles représentant des applications. Ils pourront être écrits sans qu'on ait à se soucier le moins du monde des éventuelles opérations de reconfiguration ultérieures, puisque les propriétés concernant l'assemblage et la hiérarchie des éléments constituant le métamodèle seront séparées des propriétés réflexives. Une fois l'application modélisée grâce au DSMM en exploitation, il sera possible de déduire les

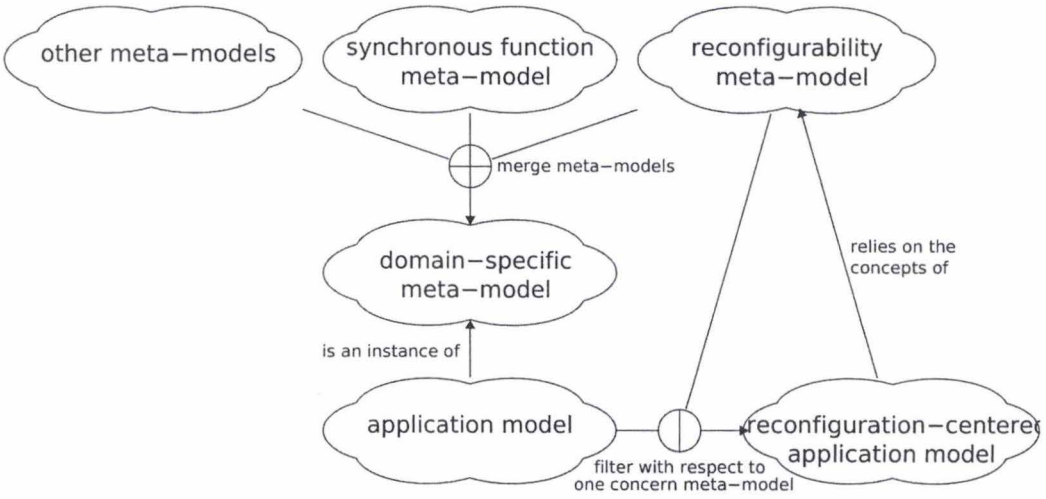


FIG. 4.2 – Une autre approche de la reconfiguration dynamique. Les métamodèles du niveau supérieur représentent différents aspects de la modélisation, tels que les différents concepts et leurs relations. Le second niveau est spécifique à chaque domaine d'application : il reprend les aspects de modélisation spécifiques à un domaine particulier. Ce DSMM permettra d'écrire des modèles représentant une partie du monde réel, c'est-à-dire une partie du domaine d'application visé. A partir de ces modèles, on pourra déduire directement l'ensemble des opérations de reconfiguration autorisées.

opérations de reconfiguration que l'application autorise. Ce qui est illustré dans la bulle *reconfiguration-centered application model* de la figure 4.2.

Les opérations « courantes » du modèle représentant l'application sont donc parfaitement séparées des opérations de reconfiguration. En effet, dans la plupart des systèmes réflexifs, toutes ces propriétés sont mises sur le même plan. Le succès actuel des approches DSMM explique en partie le besoin de séparer les nombreuses préoccupations dans le design des modèles. L'approche décrite dans ce chapitre tente de montrer les avantages de la séparation des aspects transversaux d'un métamodèle, en l'occurrence les concepts relatifs à l'observabilité et la reconfiguration d'une application, des autres propriétés utiles à la modélisation.

Dans le contexte des logiciels de vol, il sera même possible d'envoyer l'exécutable nécessaire pour accomplir la reconfiguration juste pour le temps nécessaire. Une fois les opérations effectuées, l'exécutable permettant la reconfiguration sera déchargé. Ce besoin correspond précisément au principe de stratification évoqué par [3].

La figure 4.2 illustre l'approche. Les différents métamodèles du niveau supérieur regroupant différents aspects de modélisation (dont le *synchronous function meta-model* dans le cas particulier du projet SP@CIFY) fusionnent pour donner un *domain-specific meta-model*. Ce DSMM permettra de modéliser des applications en utilisant le vocabulaire spécifique du domaine visé. Plus tard, les opérations de reconfiguration pourront être déduites des modèles d'applications.

### 4.2.1 Métamodèles

Nous introduisons ici les métamodèles du niveau supérieur de la figure 4.2. Ils expriment chacun une propriété de modélisation.

Le métamodèle **noyau** (figure 4.3) contient les concepts de base. Les métamodèles **assemblage** (figure 4.4) et **hiérarchie** (figure 4.5) décrivent les propriétés structurelles, c'est-à-dire la façon dont on peut associer les différents éléments des modèles. Le métamodèle d'assemblage introduit deux sous-types d'unités architecturales : les blocs et les connecteurs, ainsi que leurs relations. Le métamodèle de hiérarchie décrit comment une instance peut contenir des sous-instances ou être elle-même contenue dans une instance parente.

Enfin, le métamodèle d'observabilité et de reconfiguration (figure 4.6) introduit les capacités réflexives. Le système basé sur ce métamodèle pourra inspecter son architecture et contiendra un ensemble d'opérations permettant d'effectuer des reconfigurations.

Précisons que l'emploi du terme métamodèle est ici un peu problématique. En effet, ces métamodèles se situent à un niveau d'abstraction au délà du DSMM, qui est lui-même, par définition, un métamodèle. Pour être plus précis, il serait sans doute plus juste de les appeler les métamétamodèles.

#### Noyau

Dans la figure 4.3, nous voyons le métamodèle central, très général, autour duquel se greffent les concepts décrits dans les autres métamodèles. Le concept central est celui d'**unité architecturale** (*Unit*), qui définit les éléments principaux d'un modèle. Le **type** décrit les moyens d'interaction avec une unité. L'**implémentation** quant à elle contient le comportement. Une implémentation agit sur l'état interne d'une instance. Les opérations de

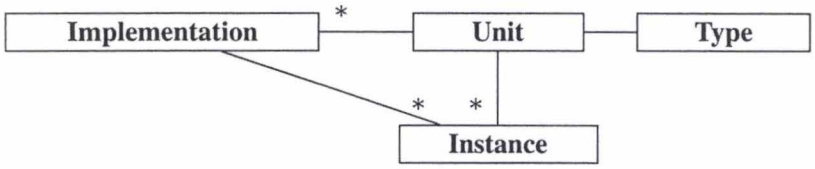


FIG. 4.3 – Le métamodèle noyau est à la base de l'approche. Tous les autres métamodèles s'y rattachent. L'unité architecturale est le concept principal. L'implémentation décrit les différents comportements des instances et le type décrit les capacités d'interactions. L'instance représente l'unité à l'exécution. Par défaut, les cardinalités sont égales à 1.

reconfiguration consisteront par exemple à charger de nouvelles implémentations (donc à insérer de nouveaux comportements) qui n'étaient pas prévues lors du design de l'application. Une unité architecturale peut posséder plusieurs implémentations. L'implémentation liée à une instance est sélectionnée lors de sa création.

Dans la programmation orientée-objet par exemple, l'unité architecturale s'identifie à la classe, tandis que les **instances**, qui représentent l'unité à l'exécution et encapsulent une état interne, sont les objets. Le type, qui décrit les capacités d'interaction de l'unité, correspond à l'ensemble des signatures des méthodes et des attributs et l'implémentation correspond aux corps des méthodes.

L'approche est assez générique pour s'adapter à une grande variété de paradigmes. Nous avons donné l'exemple de l'orienté-objet mais nous pouvons aussi l'appliquer à la programmation fonctionnelle : l'unité est la fonction, le type est la signature de la fonction, l'implémentation est le corps de la fonction et les instances correspondent aux fermetures.

Remarquons que dans certains paradigmes de modélisation, ces quatre concepts ne font qu'un. Dans les approches orientées-objet, l'implémentation (unique !) et le type sont intrinsèquement liés à la classe : la classe, ses méthodes et l'implémentation de ses méthodes ne font qu'un.

### Assemblage

Le schéma de la figure 4.4 se base sur une distinction entre des **blocs** et des **connecteurs**. Les blocs sont assez similaires à ce qu'on appelle « composants » ou « objets » dans certains paradigmes, tout se situant à un niveau d'abstraction plus élevé.

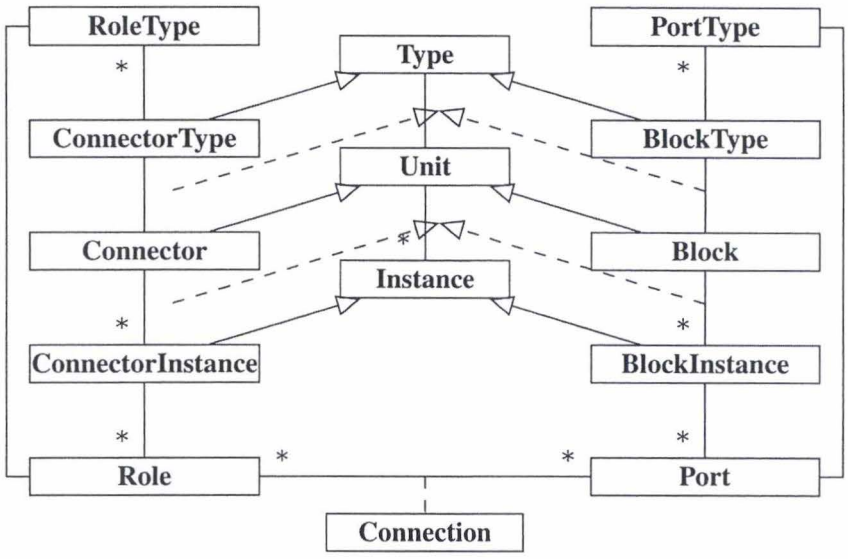


FIG. 4.4 – Métamodèle d'assemblage

Les connecteurs, dans les modèles les plus simples, existent sous la forme de simples liens entre deux blocs. Dans des modèles plus évolués (tels que les diagrammes ERA), ces liens pourront être augmentés de cardinalités ou d'attributs. Dans des DSMM encore plus évolués, ces connecteurs pourront être encore un peu plus sophistiqués, et implémenter un comportement complexe.

Ces notions de blocs et de connecteurs spécialisent tout deux le concept d'unité architecturale, ainsi que les concepts s'y référant (type, instance, etc.) Le lien unissant une instance de bloc (respectivement de connecteur) nécessitent d'introduire le nouveau concept de **port** (respectivement de **role**). L'association entre ces deux points est réifiée par la notion de **connexion**.

### Hiérarchie

Le métamodèle de la figure 4.5 introduit les concepts permettant d'exprimer des imbrications entre instances. Toujours en suivant l'exemple de l'orienté-objet, on peut évoquer le concept d'*inner class*. Une instance pourra contenir d'autres instances qui ne seront pas accessibles de l'extérieur.

Le métamodèle est construit sur une structure d'arbre récursive, augmentée du concept d'instance **composite**. Une instance composite contient plusieurs sous-instances content. Une `ContentInstance` est contenue, encapsulée

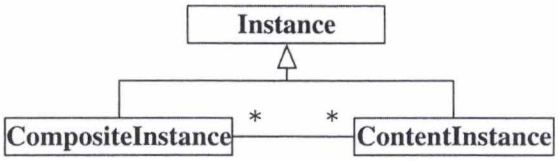


FIG. 4.5 – Métamodèle hiérarchique

dans au moins une autre instance. Une instance peut donc être tout à la fois composite (composée de plusieurs instances) et content (contenue dans une ou plusieurs instances).

Ce pattern permet d'inclure des modèles autorisant des sous-composants partagés entre plusieurs composants parents comme cela est autorisé dans Fractal. Dans une structure d'imbrication classique, qui ne permettrait pas les composants partagés, la cardinalité du côté de la `ContentInstance` devrait être égale à 1. Dans ce cas, un composant content ne serait contenu que dans un et un seul composant composé. Nous aurions alors une structure d'arbre simple.

Mais ces métamodèles tendent à être assez génériques pour permettre tous les cas. On pourra tout à fait décider, lors de la création du DSMM, s'il autorise ou pas l'usage des composants partagés. Certains DSMM pourront même interdire totalement l'usage de composants hiérarchisés, et aboutir ainsi à un modèle tout à fait plat, comme dans le modèle de composants de CORBA.

### Observabilité et reconfiguration

Les précédents métamodèles décrivent les propriétés essentielles à la modélisation d'applications et sont assez généraux pour permettre d'engendrer une grande variété de DSMM. Jusque là, nous avons fait abstraction des capacités de réflexivité et n'avons détaillé que les métamodèles utiles au design d'applications. Or, le but de l'approche est précisément de créer des DSMM dans lesquels on puisse écrire des modèles d'application capables de s'observer et de se reconfigurer.

Le figure 4.6 se base sur le métamodèle noyau (*core concepts*). L'**observabilité** correspond à la « lecture », à l'introspection de l'application ; tandis que la reconfiguration correspond à l'« écriture » dans l'application. Un système observable est capable de renseigner un certain nombre de métainformations sur

le système. On peut par exemple obtenir la liste des attributs d'une instance, tout comme on peut, en Java, obtenir la liste des méthodes d'un objet.

Au niveau de la reconfiguration, on distingue des unités architecturales de base, qui contiennent un type, un ensemble d'implémentations et un ensemble d'instances qui la représentent à l'exécution ; et les unités architecturales reconfigurables, qui permettent en plus de créer de nouvelles instances et d'en supprimer, de charger et de décharger des implémentations. Le type peut lui aussi être reconfigurable et permettre par exemple l'ajout et la suppression de type d'attributs. Une instance reconfigurable peut être mise en marche ou arrêtée. Elle peut également changer d'implémentation, parmi celles proposées par l'unité architecturale.

Si l'on désire par exemple changer le comportement d'une instance en exécution, on procède comme suit :

1. Rendre l'unité architecturale reconfigurable si elle ne l'est pas
2. Charger le nouveau comportement dont on a besoin avec l'opération `ReconfigurableUnit.loadImplementation()`
3. Rendre l'instance reconfigurable si elle ne l'est pas
4. Arrêter l'instance avec l'opération `ReconfigurableInstance.stop()` si la modification le nécessite
5. Changer l'implémentation de l'instance reconfigurable avec l'opération `ReconfigurableInstance.setImplementation()`
6. Redémarrer l'instance avec l'opération `ReconfigurableInstance.start()` si elle a été arrêtée
7. Rendre l'unité architecturale et l'instance à nouveau non-reconfigurable si cela est nécessaire

Notons que tous les éléments d'une application ne sont pas *obligatoirement* reconfigurables. Ils pourront par exemple le devenir pendant le temps nécessaire pour réaliser une reconfiguration, et redevenir ensuite des éléments non-reconfigurables. Pour certains éléments, on pourrait décider lors du design de l'application que quoi qu'il arrive, il sera impossible de les reconfigurer. En terme d'ouverture, nous avons ici la possibilité de construire une gamme de systèmes dont certains seront fermés et rigides et d'autres totalement ouverts et flexibles.

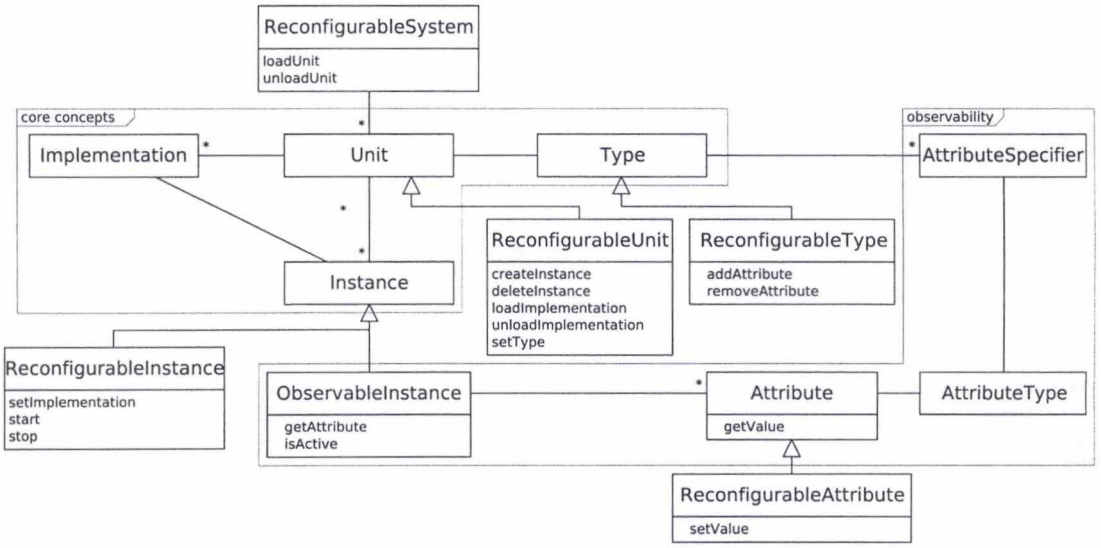


FIG. 4.6 – Métamodèle de reconfiguration et d'observabilité

## 4.3 Expériences/Implémentations

Nous allons maintenant pouvoir faire correspondre les concepts concrets de Fractal (décrits à la section 3.4, repris dans le métamodèle de la figure 4.7) aux propriétés décrites dans les différents métamodèles. Voir [5] pour plus d'informations.

- Par rapport au métamodèle de base, décrit dans la figure 4.3 (p. 70), un composant Fractal équivaut à une instance, un type de composant Fractal équivaut à un type, et un comportement correspond à une implémentation.

- Par rapport au métamodèle d'assemblage, décrit dans la figure 4.4 (p. 71), une composant Fractal correspond une instance de bloc, un type de composant Fractal correspond à un type de bloc, une interface Fractal est un port et une type d'interface est un type de port.

- Par rapport au métamodèle hiérarchique, décrit dans la figure 4.5 (p. 72), un composant Fractal une instance composite et content. Remarquons que le métamodèle, de par sa généricité, autorise le concept des composants partagés. Plus précisément, c'est la cardinalité du côté de l'instance content qui indique qu'une même instance peut-être conte-

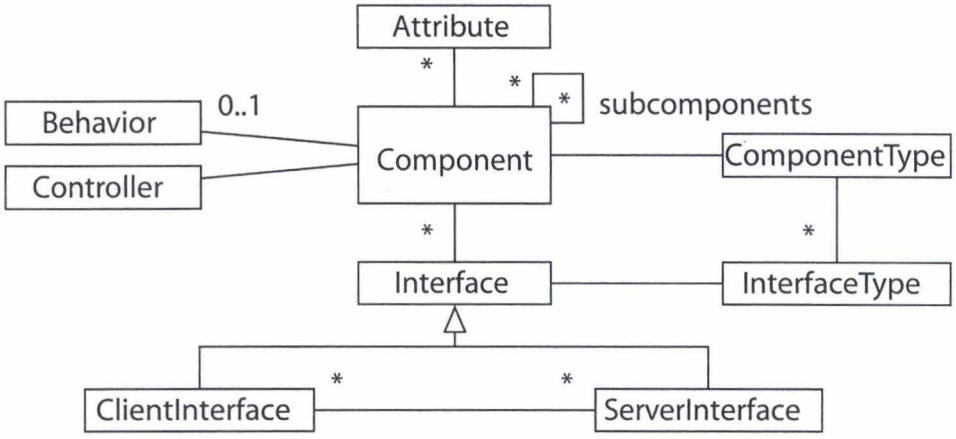


FIG. 4.7 – Un métamodèle possible pour Fractal. Source : [5]

nue dans plusieurs instances composites.

- Enfin, par rapport au métamodèle d'observabilité et de reconfiguration, décrit dans la figure 4.6 (p. 74), un composant Fractal est une instance à la fois observable et reconfigurable. Les types de composants dans Fractal n'ont pas besoin d'être des `ReconfigurableType` puisque comme nous l'avons vu, les types de composants en Fractal ne peuvent être modifiés.

# Chapitre 5

# Discussion

L'approche précédemment décrite permet de mieux séparer les différentes propriétés de modélisation que dans la plupart des modèles architecturaux existants. Les propriétés essentielles au design peuvent être isolées des autres propriétés, ainsi que les concepts relatifs à l'introspection et à la reconfiguration de l'application.

Par ailleurs, la généricité et l'extensibilité des métamodèles autorisent l'insertion d'éléments spécifiques à un domaine d'application donné. Des modèles propres au domaine peuvent alors être écrits, sans qu'on ait à se soucier des opérations de maintenance ultérieures. L'approche offre la possibilité de créer une grande variété de DSMM, et de déduire automatiquement les opérations de reconfiguration dynamique qui y sont autorisées.

L'approche généralise également, en s'élevant à un niveau d'abstraction au dessus des modèles orientés-composant, la granularité. Tous les éléments du modèle peuvent être reconfigurables. L'exemple de reconfiguration Fractal (p. 47), qui consistait à transformer l'implémentation d'un composant, nécessitait de supprimer l'ancienne version du composant et de la remplacer par un autre qui possédait la nouvelle implémentation. On peut ici effectuer la même opération en une seule étape, en invoquant sur l'instance l'opération `setImplementation()`.

Précisons néanmoins que s'il existe plusieurs bonnes raisons pour lesquelles nous désirons séparer les fonctionnalités de base d'un système et les fonctionnalités utiles à la reconfiguration, décider si une opération donnée appartient à l'une ou l'autre catégorie est parfois compliqué. Dans la figure 4.6 (p. 74) en effet, l'opération permettant de lire un attribut se situe dans le cadre

de l'observabilité et l'opération permettant de le modifier se situe au niveau de la reconfiguration. Comme [3] en défend l'idée, les méta-opérations devraient se trouver un méta-niveau. Or, la lecture et l'écriture d'un attribut pourraient être considérées comme des opérations du niveau de base. Les figures représentant les métamodèles qui schématisent les différentes propriétés (assemblage, hiérarchie, observabilité et reconfiguration) sont ici à titre d'exemple. L'approche en effet ne consiste guère en un ensemble rigide d'opérations. Elle a plutôt pour objectif de montrer le gain de présenter séparément les différentes propriétés utilisées pour décrire des applications.

On peut analyser l'approche selon les mêmes caractéristiques d'analyse qu'auparavant :

**Granularité :** La granularité est généralisée à tous les éléments du modèle : ajout et suppression d'instances, de types, d'implémentation et d'unités architecturales elles-mêmes.

**Type de modification :** L'approche s'inspire essentiellement des architectures orientées-composant (tout en étant plus abstraite et plus générale) et les modifications structurelles sont donc autorisées : ajout et suppression d'instances, etc. On peut également procéder à des modifications comportementales, en changeant l'implémentation d'une instance. L'ajout, la suppression ou la modification de type correspondent également à des modifications de type structurel. L'approche autorise aussi l'ajout et la suppression d'unités architecturales. Ceci implique que de nouveaux concepts peuvent être insérés dans une application. Ces modifications ne sont ni structurelles ni comportementales, mais devraient plutôt être considérées dans une troisième catégorie. Nous pouvons parler ici de modification conceptuelle ou ontologique.

**Ouverture :** Comme déjà évoqué plus haut, l'approche est assez générique pour inclure plusieurs niveaux d'ouvertures entre différents DSMM ou même au sein d'un même DSMM. C'est précisément à ce niveau, lors du choix des concepts qui serviront dans le DSMM que l'on décide si les applications seront totalement fermées et rigides ou si, au contraire, tous les éléments seront reconfigurables. Des politiques intermédiaires sont envisageables : certains éléments du modèles pourraient demeurés inchangés, tandis que d'autres seraient plus flexibles.

**Gestion de l'état :** Le problème de la gestion de l'état n'est pas abordé dans l'approche. Néanmoins, on peut s'inspirer des différentes techniques

prévues dans les solutions étudiées dans l'état de l'art. Nous abordons la question du transfert d'état de plus loin dans le document.

Néanmoins, si l'approche développée ici permet de supprimer des éléments, elle ne répond pas à la question de savoir l'effet sur les instances d'une suppression de type ou d'implémentation. Nous pouvons néanmoins envisager deux stratégies qui, sous certaines réserves, ressemblent aux modes de gestion de l'intégrité référentielle en base de données. Que faire en effet si l'on veut supprimer et décharger une implémentation d'un système alors que des instances ? Deux stratégies s'offrent à nous : (i) la stratégie en cascade, qui supprime automatiquement toutes les instances qui utilisent cette implémentation[1], ou (ii) la stratégie de blocage, qui empêche de supprimer l'implémentation et éventuellement renvoie un message d'erreur. Pour certaines opérations, une troisième stratégie s'offre à nous : celle du *versionning*. Si l'on ajoute des attributs à un type, les instances déjà crées auront le numéro de version $i$ ; après cette modification, les instances auront le numéro $i + 1$ et ainsi de suite. Ce problème nous amène à la question du transfert d'état.

## 5.1 Transfert d'état

Nous avons abordé avec PJama le problème de conversions des instances et nous venons d'envisager des possibilités de réponse lors d'une modification de type ou une suppression d'implémentation. Que faire si l'on supprime ou/et ajoute des attributs à un type ? Comme dans la conversion des instances, les attributs supprimés des types seraient aussitôt supprimés des instances et perdraient leur valeur tandis que les nouveaux attributs seraient initialisés. La conversion personnalisée (*custom conversion*) implémentée pour Java permet au programmeur d'insérer du code qui décrive la transformation.

Ketfi [27] nous propose cependant de penser le problème d'une façon plus abstraite. Il distingue l'état « technique » d'une instance, c'est-à-dire la valeur de tous ses attributs et l'état abstrait, celui qui reflète la sémantique de l'instance (cf. figure 5.1). L'idée développée ici distingue les deux types d'état et maintient la relation qui existe entre eux. L'outil permet de sélectionner le sous-ensemble des attributs de l'instance qui constituent effectivement l'état sémantique. Cet état peut alors être échangés entre plusieurs instances qui partagent au moins ce sous-ensemble d'attributs. Les opérations `getState()` et `setState()` sont générées automatiquement.

[1] Notons que [27], dans son modèle Dyva, impose que la suppression d'un type de composant entraîne automatiquement la suppression de ses instances

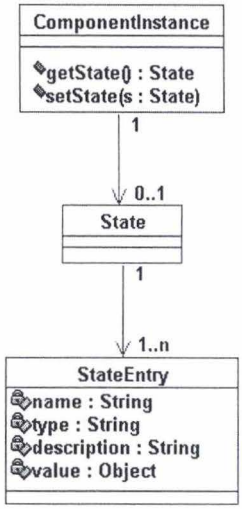


FIG. 5.1 – Modèle abstrait de l'état. Source : [27]

## 5.2 Langages de reconfiguration

Jusqu'à présent, nous n'avons considéré les opérations de reconfiguration que comme des primitives. Or, on pourrait envisager l'utilisation de scripts de reconfiguration. Pour plus de sécurité et d'intégrité, on pourrait souhaiter une technologie de reconfiguration transactionnelle. Par exemple, si l'on veut modifier l'implémentation d'un composant qui encrypte des requêtes et d'un composant qui les décrypte lors de la réception, il est nécessaire que les deux soient transformés atomiquement.

Il existe pour Fractal deux langages qui répondent à ces besoins [14, 15]. FPath, tout d'abord, est une langage de navigation pour les architectures Fractal. Il utilise les opérations d'introspection. Concrètement, une requête FPath adressée à une architecture Fractal renvoie un ensemble d'éléments architecturaux tels que des composants ou des interfaces. Par exemple, l'expression suivante renvoie les sous-composants du composant `c` qui fournissent l'interface `aService` :

```
$c/child::*/interface::aService
```

Le second exemple renvoie les interfaces clients des sous-composants du composant `c` qui ne sont pas connectées :

```
$c/interface::*[required(.)][not(bound(.))];
```

Il s'agit d'un langage déclaratif qui permet d'écrire des expressions brèves dont les équivalents, s'ils étaient implémentés via les APIs des contrôleurs, nécessiteraient plusieurs boucles imbriquées. FScript de son côté permet la reconfiguration. FPath est inclus dans FScript comme un sous-ensemble, c'est-à-dire que les scripts FScript peuvent inclure des expressions FPath. La fonction suivante (qu'on appelle les « actions » en FScript) explore les sous-composants du composant comp passé en paramètre et connecte entre elles les interfaces clients et serveurs qui sont compatibles.

```
action auto-bind(comp) = {
   // Selects the interfaces to connect
   clients := $comp/interface::*[required(.)][not(bound(.))];
   foreach itf in $clients do {
      // Search for candidates compatible interfaces
      candidates := $comp/sibling::*/interface::*[compatible?($itf, .)];
      if (not(empty?($candidates))) {
         // Connect one of these candidates
         bind($itf, one-of($candidates));
      }
   }
   return empty?($comp/interface::*[required(.)][not(bound(.))]);
}
```

# Chapitre 6

# Conclusion

Après avoir brièvement introduit les concepts qui sous-tendent les architectures matérielle et logicielle des satellites, nous avons parcouru et analysé diverses solutions intégrant des mécanismes permettant la reconfiguration dynamique, développées pour répondre à divers besoins. Arachne a été créé pour répondre au besoin précis d'adaptabilité des caches web, qui devaient être maintenus pour être adaptés aux nouvelles utilisations des protocoles d'Internet, tandis que Fractal a été pensé dans le but de fournir un modèle de composants utilisable pour une très large variété d'applications, du middleware au système d'exploitation.

Nous avons montré les inconvénients des solutions les moins abstraites et les avantages des solutions qui prévoyaient à l'avance leurs mécanismes de reconfiguration. Nous avons également souligné l'importance de laisser la possibilité de pouvoir supprimer des éléments, en analysant les difficultés que cela impliquait. Nous avons montré les avantages de modèles dont le couplage entre les éléments est limité.

Enfin, la synthèse des problèmes tant conceptuels que techniques rencontrés dans les différentes solutions analysées dans l'état de l'art, des limites inhérentes aux différents outils, nous a permis de penser une nouvelle approche, indépendante des contraintes propres à certains langages. Cette dernière approche combine plusieurs avantages des différentes solutions. Comme pour Arachne et PJama, les opérations de reconfiguration sont séparées des opérations métiers ; comme pour Fractal, on définit un modèle de composants abstrait, extensible, indépendant du langage d'implémentation.

Quelques questions demeurent en suspend et mériteraient de plus amples développements comme le transfert d'état ou la conversion d'instances (à laquelle PJama nous apporte quelques pistes intéressantes).

Le problème de l'auto-reconfiguration et du *self-healing* aurait pu faire l'objet d'un chapitre supplémentaire. En effet, tout au long de notre analyse, nous avons fait l'hypothèse implicite que les opérations de reconfiguration étaient effectuées par une équipe de développeurs chargées de la maintenance, de la correction de bugs, etc. Or, on peut penser à des logiciels auto-adaptables. Ces problématiques impliquent l'ajout de moyens de détection des erreurs de comportement et/ou les moyens de laisser le système faire lui-même le choix de sa nouvelle configuration. Le lecteur intéressé peut notamment se référer à [16, 21, 36] pour plus d'informations.

Enfin, le concept de « reconfiguration transactionnelle » est envisagé brièvement dans différents contextes : les component frameworks d'OpenORB abordent l'idée d'une cohérence entre plusieurs opérations de reconfiguration primitives et les langages tels que FScript développent l'idée de scripts de reconfiguration respectant les propriétés ACID.

# Annexe

## Grammaire du langage d'aspect Arachne

Source : [36]

| | | |
|---|---|---|
| *aspect-library* | ::= | *aspect* \| *aspect aspect-library*$^+$ \| *C-compound-instruction aspect-library*$^+$ \| *require-directive aspect-library*$^+$ |
| *aspect* | ::= | *aspect-name* "[:" *pointcut-advice* ":]" |
| *pointcut-advice* | ::= | *function-call* "[:" *pointcut-advice* ":]" \| *function-call* "[:" *advice* ":]" \| *global-var-access* "[:" *advice* ":]" |
| *function-call* | ::= | *type identifier-or-star* "(" *params* ")" \| *type identifier-or-star* "()" |
| *params* | ::= | *type identifier* \| *params* "," *params* |
| *global-var-access* | ::= | *global-var-read* \| *global-var-write* |
| *global-var-read* | ::= | *type identifier-or-star* |
| *global-var-write* | ::= | *type identifier-or-star* "=" *identifier* |
| *advice* | ::= | *C-compound-instruction* |
| *identifier-or-star* | ::= | *identifier* \| "*" |
| *aspect-name* | ::= | *identifier* |
| *require-directive* | ::= | "*require*" *identifier* "*as*" *type* |

## API Fractal

Il s'agit ici de l'API en Java. La description détaillée de l'API peut se trouver dans [4].

```
package org.objectweb.naming;
public interface Name {
  NamingContext getNamingContext ();
  byte[] encode () throws NamingException;
}
public interface NamingContext {
  Name export (Object o, Object hints) throws NamingException;
  Name decode (byte[] b) throws NamingException;
}
public interface Binder extends NamingContext {
  Object bind (Name n, Object hints) throws NamingException;
}
public class NamingException extends Exception {
```

```
  public NamingException (String msg) { super(msg); }
}
package org.objectweb.fractal.api;
import org.objectweb.fractal.api.factory.InstantiationException;
public interface Component {
  Type getFcType ();
  Object[] getFcInterfaces ();
  Object getFcInterface (String interfaceName) throws NoSuchInterfaceException;
}
public interface Interface {
  Component getFcItfOwner ();
  String getFcItfName ();
  Type getFcItfType ();
  boolean isFcInternalItf ();
}
public interface Type {
  boolean isFcSubTypeOf (Type type);
}
public class Fractal {
  public static Component getBootstrapComponent ()
    throws InstantiationException;
}
public class NoSuchInterfaceException extends Exception {
  public NoSuchInterfaceException (String itfName) { super(itfName); }
}
package org.objectweb.fractal.api.control;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.NoSuchInterfaceException;
public interface AttributeController { }
public interface BindingController {
  String[] listFc ();
  Object lookupFc (String clientItfName) throws NoSuchInterfaceException;
  void bindFc (String clientItfName, Object serverItf) throws
    NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException,
  void unbindFc (String clientItfName) throws
    NoSuchInterfaceException, IllegalBindingException, IllegalLifeCycleException,
}
public interface ContentController {
  Object[] getFcInternalInterfaces ();
  Object getFcInternalInterface (String interfaceName)
    throws NoSuchInterfaceException;
  Component[] getFcSubComponents ();
  void addFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
  void removeFcSubComponent (Component subComponent)
    throws IllegalContentException, IllegalLifeCycleException;
}
public interface SuperController {
  Component[] getFcSuperComponents ();
}
public interface LifeCycleController {
  String getFcState ();
  void startFc () throws IllegalLifeCycleException;
  void stopFc () throws IllegalLifeCycleException;
}
public interface NameController {
  String getFcName ();
  void setFcName (String name);
}
```

```
public class IllegalBindingException extends Exception {
  public IllegalBindingException (String msg) { super(msg); }
}
public class IllegalContentException extends Exception {
  public IllegalContentException (String msg) { super(msg); }
}
public class IllegalLifeCycleException extends Exception {
  public IllegalLifeCycleException (String msg) { super(msg); }
}
package org.objectweb.fractal.api.factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.Type;
public interface Factory {
  Type getFcInstanceType ();
  Object getFcControllerDesc ();
  Object getFcContentDesc ();
  Component newFcInstance () throws InstantiationException;
}
public interface GenericFactory {
  Component newFcInstance (Type type, Object controllerDesc, Object contentDesc)
    throws InstantiationException;
}
public class InstantiationException extends Exception {
  public InstantiationException (String msg) { super(msg); }
}
package org.objectweb.fractal.api.type;
import org.objectweb.fractal.api.NoSuchInterfaceException;
public interface ComponentType extends org.objectweb.fractal.api.Type {
  InterfaceType[] getFcInterfaceTypes ();
  InterfaceType getFcInterfaceType (String name) throws NoSuchInterfaceException,
}
public interface InterfaceType extends org.objectweb.fractal.api.Type {
  String getFcItfName ();
  String getFcItfSignature ();
  boolean isFcClientItf ();
  boolean isFcOptionalItf ();
  boolean isFcCollectionItf ();
}
public interface TypeFactory {
  InterfaceType createFcItfType (
    String name, String signature,
    boolean isClient, boolean isOptional,
    boolean isCollection)
      throws org.objectweb.fractal.api.factory.InstantiationException;
  ComponentType createFcType (InterfaceType[] interfaceTypes)
    throws org.objectweb.fractal.api.factory.InstantiationException;
}
```

# API OpenCOM

Il s'agit ici de l'API en C++ [11]. Une version Java, quelque peu différente, peut être consultée [25].

```
interface OCM_IOCM : IUnknown
```

```
{
   HRESULT createInstance(
      [in] REFCLSID rclsid,
      [out] IUnknown **ppIUnknown,
      [in, string] const unsigned char *name);
   HRESULT deleteInstance(
      [in] IUnknown *pIUnknown,
      [out] OCM_IReceptacle *ppOCM_IRecps[],
      [out] int *pcElems);
   HRESULT connect(
      [in] IUnknown *pIUnkSource,
      [in] IUnknown *pIUnkSink,
      [in] REFIID iid,
      [in] OCM_RecpID_t recpID,
      [out] OCM_ConnID_t *pConnID);
   HRESULT disconnect(
      [in] OCM_ConnID_t connID,
      [out] OCM_IReceptacle **ppOCM_IRecp);
   HRESULT getConnectionInfo(
      [in] OCM_ConnID_t connID,
      [out] OCM_ConnInfo_t **ppConnInfo);
   HRESULT freeConnectionInfo(
      [in] OCM_ConnInfo_t *pConnInfo);
   HRESULT enumComponents(
      [out] IUnknown* *ppComps[],
      [out] int *pcElems);
   HRESULT getComponentName(
      [in] IUnknown *pIUnknown,
      [out] unsigned char **ppName);
   HRESULT getComponentPIUnknown(
      [in, string] const unsigned char *name,
      [out] IUnknown **ppIUnknown);
   HRESULT getComponentCLSID(
      [in] IUnknown *pIUnknown,
      [out] CLSID *pclsid);
};

interface OCM_ILifeCycle : IUnknown
{
   HRESULT startup([in] IOCM *pIOCM);
   HRESULT shutdown(void);
}

interface OCM_IReceptacles : IUnknown
{
   HRESULT connect(
      [in] void *pSinkIntf,
      [in] REFIID riid,
      [in] OCM_ConnID_t provConnID,
      [in] OCM_RecpID_t recpID);
   HRESULT disconnect(
      [in] REFIID riid,
      [in] OCM_ConnID_t connID,
      [in] OCM_RecpID_t recpID);
};

interface IMetaArchitecture : IUnknown
{
   HRESULT enumConnsToIntf(
```

```
      [in] REFIID riid,
      [out] OCM_ConnID_t *ppConnsToIntf[]
      [out] int *pcElems);
   HRESULT enumConnsFromRecp(
      [in] REFIID riid,
      [out] OCM_ConnID_t *ppConnsFromRecp[]
      [out] int *pcElems);
};

interface IMetaInterface : IUnknown
{
   HRESULT enumRecps(
      [out] OCM_RecpMetaInfo_t *ppRecpMetaInfo[],
      [out] int *pcElems);
   HRESULT enumIntfs(
      [out] IID *ppIntf[],
      [out] int *pcElems);
};


interface IMetaInterception : IUnknown
{
   HRESULT createInterceptor(
      [in] OCM_InterceptorType_t intcpType,
      [in] REFIID riid,
      [out] OCM_IInterceptor **ppOCM_IInterceptor);
   HRESULT deleteInterceptor([in] OCM_InterceptorType_t intcpType,
      [in] OCM_IInterceptor *pOCM_IInterceptor);
};

interface IInterceptor
{
   HRESULT addPreMethod(
      [in] const unsigned char *DLLName,
      [in] const unsigned char *methodName);
   HRESULT delPreMethod([in] const unsigned char *methodName);
   HRESULT addPostMethod(
      [in] const unsigned char *DLLName,
      [in] const unsigned char *methodName);
   HRESULT delPostMethod([in] const unsigned char *methodName);
   HRESULT viewPreMethods([out] unsigned char *methodNames[]);
   HRESULT viewPostMethods([out] unsigned char *methodNames[]);
   HRESULT deleteInterceptor(void);
};
```

# Bibliographie


[1] Openorb - a marriage of three technologies. Technical report, Lancaster University, Computing Departement. `http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/meta.php`.

[2] Gordon Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. Reflective middleware : The design and implementation of openorb 2. *Metalevel Architectures and Separation of Crosscutting Concerns. Lecture Notes in Computer Science*, 2192 :268–269, 2001.

[3] Gilad Bracha and David Ungar. Mirrors : design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04 : Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM.

[4] Eric Bruneton, Thierry Coupaye, and Jean Bernard Stefani. The fractal component model. Technical report, The ObjectWeb Consortium, 2004. `http://fractal.objectweb.org/specification/index.html`.

[5] Jérémy Buisson and Fabien Dagnat. Experiments with fractal on modular reflection. In *SERA'08 : 6th international conference on software engineering research, management and applications*, pages 20–22, 2008.

[6] David Chemouil. Ingénierie des modèles et méthodes formelles intégrées pour le développement des logiciels de vol spatiaux - projet de recherche et développement à caractère exploratoire. *Dossier de soumission au programme Technologies logicielles de l'Agence nationale de la recherche en réponse à l'appel à projets ANR/DMR/TL/AAP/220306-01*, décembre 2006.

[7] David Chemouil. Spa1. définition des besoins. *Projet ANR/RNTL SPaCIFY (réf. ANR 06 TLOG 27)*, 2007.

[8] Yan Chen. Aspect-oriented programming (aop) : Dynamic weaving for c++. Master's thesis, Vrije Universiteit Brussel and Ecole des Mines de Nantes, 2003.

[9] ObjectWeb Consortium. C language support : Cecilia. `http://fractal.objectweb.org/c.html`.

[10] Geoff Coulson, Gordon Blair, Michael Clarkel, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg. Lecture Notes In Computer Science*, 2218 :160–178, 2001.

[11] Geoff Coulson, Gordon Blair, Michael Clarkel, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15 :109–126, 2002.

[12] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. *Proceedings of IASTED Software Engineering and Applications (SEA 04)*, 2004.

[13] Geoff Coulson, Paul Grace, Gordon Blair, David Duce, Chris Cooper, and Musbah Sagar. A middleware approach for pervasive grid environments. *Workshop on Ubiquitous Computing and e-Research National eScience Centre*, 2005.

[14] Pierre-Charles David. Fpath and fscript reference manual. Technical report, The ObjectWeb Consortium, novembre 2007.

[15] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. *Proceedings of the 5th Fractal Workshop at ECOOP 2006, Nancy, France*, july 2006.

[16] Ada Diaconescu and John Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 44–53, New York, NY, USA, 2005. ACM.

[17] Mikhail Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. PhD thesis, University of Glasgow, mars 2001.

[18] Mikhail Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*. Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303 US, september 2001.

[19] Mikhail Dmitriev and M. Atkinson. Evolutionary data conversion in the pjama persistent language. *Proceedings of the Workshop on Object-Oriented Technology. Lecture Notes In Computer Science*, 1743 :211–212, 1999.

[20] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05 : Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM.

[21] Xabier Elkorobarrutia, Alberto Izagirre, Goiuria Sagardui, Xabier Elkorobarrutia, Alberto Izagirre, and Goiuria Sagardui. A self-healing mechanism for state machine based components. *Proceedings of the 1st International Conference on Ubiquitous Computing : Applications, Technology and Social Issues, Alcala de Henares*, 2006.

[22] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *ICSE '76 : Proceedings of the 2nd international conference on Software engineering*, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[23] Thomas Fritz, Marc Ségura, Mario Südholt, Egon Wuchner, and Jean-Marc Menaud. An application of dynamic aop to medical image generation. *Dynamic Aspects Workshop (DAW05) in conjonction with the 4th international conference on Aspect-oriented software development, ACM Press*, mars 2005.

[24] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[25] Next Generation Middleware Group. Opencom : Java versus c++. Technical report, Lancaster University, Computing Department, 2003.

[26] Ackbar Joolia, Geoff Coulson, Gordon Blair, Antonio Tadau Gomes, Kevin Lee, and Jo Ueyama. Flexible programmable networking : A reflective, component-based approach. *PGNet*, 2003.

[27] Abdelmadjid Ketfi. *Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels*. PhD thesis, Université Joseph Fourier de Grenoble, décembre 2004.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, second edition, 1999.

[30] Nicolas Loriant, Marc Ségura-Devillechaise, , and Jean-Marc Menaud. Un bac à sable juste à temps, correctifs ciblés et injection à chaud. *5ème Conférence Française sur les Systèmes d'Exploitation (CFSE'05)*, pages 27–38, mars 2005.

[31] Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Des correctifs de sécurité à la mise à jour - audit déploiement distribué et injection à chaud. *DECOR'2004, 1ère conférence Francophone sur le déploiement et la (re)configuration de Logiciels*, pages 65–76, octobre 2004.

[32] Xavier Olive. On board software patchs. current practices, evolutions and needs, 2008. Thales Alienia Space, EL/PE/S - On-board software RD Group.

[33] ESA-ESTEC. Requirements and Standards Division. Space engineering. ground systems and operations - telemetry and telecommand packet utilization. ecss-e-70-41a., january 2003.

[34] Marc Ségura-Devillechaise and Jean-Marc Menaud. Caching web services : Aspect orientation to the rescue. In *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, pages 42–52, London, UK, 2002. Springer-Verlag.

[35] Marc Ségura-Devillechaise and Jean-Marc Menaud. microdyner : un noyau efficace pour la tissage dynamique d'aspects sur processus natif en cours d'exécution. *Congrès LMO 2003 Langages et modèles à objets. L'Objet ISSN 1262-1137*, 9 :119–133, 2003.

[36] Marc Ségura-Devillechaise and Jean-Marc Menaud. From legacy web caches to self-adaptable web caches : an aspect-oriented solution. Technical report, École des Mines de Nantes, 2004.

[37] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect : Towards a dynamic-weaving based solution. *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD'03). ACM Press*, 2002.

[38] Marc Ségura-Devillechaise, Jean-Marc Menaud, Julia Lawall, and Gilles Muller. Extensibilité dynamique dans les caches web : une approche par aspects. *3ème Conférence Française sur les Systèmes d'Exploitation (CFSE'03)*, pages 477–487, octobre 2003.

[39] Consortium SPACIFY. Ft1.3. cas d'étude initial v0.7. *Projet ANR/RNTL SPaCIFY (réf. ANR 06 TLOG 27)*, 2007.

[40] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 1998.